LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-585

REORDERING WITH HINDSIGHT

Bradford T. Spiers

November 1993

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Reordering with Hindsight

by

Bradford T. Spiers

This technical report is based on my master's thesis by the same name.

Reordering with Hindsight

by

Bradford T. Spiers

Abstract

This report presents *the reordering technique* for parallel debugging. This technique is useful for debugging *ordering errors*, caused when actions a programmer meant to occur in a specific order occur in a different, unintended order. These errors occur because the programmer forgot necessary synchronization. Current debugging techniques require the programmer to change the program to run actions in a different order. But the lack of control in the run of a changed program can cause trouble; the error symptoms can vanish before the programmer has a chance to fix it.

The reordering technique gives the programmer the needed control to try different action orders. The key is to change not the program, but the log of a program execution. The programmer specifies a different *ordering* of actions, and the debugger changes the log to reflect this ordering. The debugger then controls the program up to and including the reordered actions. By controlling before the ordering, the program will reach these actions. By controlling during the ordering, the debugger will test the hypothesis of the programmer. In sum, this technique helps a programmer debug errors he could not fix otherwise.

This report also describes a reordering debugger called *Hindsight*. Hindsight has been used to perform experiments evaluating the reordering technique. In each experiment, a published algorithm that has an ordering error is run in different orders. These experiments demonstrate that the reordering technique can be a valuable component of a parallel debugger.

Acknowledgments

I would like to thank:

Andrew Szanton, my writing coach, who teaches as well as he writes. He has drawn me into a love affair with words, a feat I once thought impossible. I dedicate this technical report to him.

Al Davis for leading me at Hewlett Packard Laboratories. He empowered me with room to think for myself, and occasionally reined me in with his future visions. Thanks for carefully reading my thesis and guiding me. Thanks also for Sunday barbecues, Giants games, motorcycle rides, and talks about waxing skis.

Bill Weihl for advising me during the last four years. Bill has provided insightful technical guidance throughout. I have learned a lot from Bill, especially about writing.

Kelli Foster for starting to improve my writing. My meetings with her at the MIT Writing Center greatly improved the presentation of ideas in my original Master's thesis.

Paul Cosway for discussing both my thesis and his thesis numerous times. Thanks also for golfing outings, talks about management, and discussions about writing.

Tom Wagner for being his wild, yet meticulous, self. Tom was always ready for overnight bike trips to Cape Cod, outings to toy stores, trips to the movies, and talks about the bike store we will open someday.

Noel and Bob of the Vermilion, Ohio Radio Shack outlet. They willingly taught me how to program on a TRS-80 Model I in the store. I thank them for teaching an overly curious sixth-grader.

Michael McCarthy, my friend and high-school mentor. Mike will always hold a special place in my heart. He taught me not only the basics of computer science, but how to laugh at life as well.

Karen Spiers, my sister, for supporting me and for showing me an example of a busy, successful person with a balanced life.

Chris Tsien for her love, support, caring, and friendship. She has made me happier than I once thought possible.

Tom and Shirley Spiers, my parents. They have always supported me in whatever endeavor I have chosen. A son cannot ask for a better gift.

This thesis work was supported by a NDSEG fellowship and Hewlett Packard Laboratories. It was supported indirectly by DARPA contract N00014-91-J-1698, equipment grants from DEC, and grants from IBM and AT&T.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A programmer uses parallel computers for just one reason: speed. But if he strives for speed alone, his program may lack necessary synchronization. Without synchronization, *ordering errors* can occur; actions he intends to occur in one order can instead occur in a different order. Such errors, which depend on fragile timing differences, are notoriously difficult to debug. These differences arise at the start of a program run and change easily, eliminating all symptoms of an error before the programmer has a chance to fix it.

Sequential debuggers cannot help programmers fix ordering errors because these debuggers depend on a program to run the same way multiple times. Given the same input, a sequential program will always produce the same output.[1] On the other hand, a parallel program will not always produce the same output from the same input. But a parallel programmer does want two sequential debugger features: re-running a program the same way; and examining its variable values.

Replay debuggers allow a programmer to re-run and examine a previous execution of a parallel program [LMC87, MC89b, Ber91, NM92]. Using these features from sequential debuggers, a programmer can find an error caused by an incorrect action order. But this does not necessarily mean he can fix it. Some ordering errors are subtle, so the programmer is unsure how to

[1] Let us assume a sequential system that protects a programmer from sources of nondeterminism like memory allocators and interrupts.

fix them. He starts with a hypothesis of the following form: "The incorrect output is caused when action **B** occurs before action **A**."

To test such a hypothesis, the programmer must change the program. But the programmer can inadvertently hide errors while running a changed program. The underlying problem is a lack of control over action order. While testing the hypothesis described above, timing-dependent actions before actions **A** and **B** could run in a different order, changing variable values so that actions **A** and **B** would never run.

To reproduce an error-causing environment, the programmer will want to replay the part of the program before actions **A** and **B**. But changed programs cannot be replayed. The program may no longer be able to run in the order specified by the log. The programmer is stuck running a changed program that may not run the actions that he believes cause an error.

But a worse problem exists. Even when error-causing actions run in an expected order, the programmer cannot tell whether his change deserves the credit. Two other possibilities exist. First, the programmer might have incorrectly fixed the program. The erroneous actions ran correctly anyway; the delicate timings that caused an error changed to hide its symptoms. Second, the programmer might have believed that the wrong actions caused an error. In this case, the programmer might have forced the wrong actions to occur in a specific order. Meanwhile, actions really causing an error ran in a different order, causing an expected result.

The two sequential debugging features are not enough. Parallel debuggers must confront this new type of error with a new tool. Parallel debuggers must provide tools that give the programmer necessary control over action order.

The *reordering technique* gives a programmer control over action order. This debugging technique allows him to specify that concurrent actions will run in a different order. He does so by changing the event log, which describes a program run. Since the program does not change, but action order does, the debugger can replay actions up to and including the specified order. This partial replay ensures that the program runs reordered actions. As a result, the programmer can test many hypotheses about an error. And, unlike the replay technique mentioned above, the error symptoms will not disappear before the programmer can fix the

error.

In this thesis, we will learn how to use and implement the reordering technique. We will follow experiments that use a reordering debugger, Hindsight, to evaluate this technique. These experiments suggest that the reordering technique provides programmers with a needed weapon to fight the ongoing battle against ordering errors. In the past, programmers stood by, watching helplessly as ordering errors vanished at will, only to reappear at the worst possible time. Now, with the reordering technique in their arsenal, programmers can capture these errors to debug them, preventing them from hiding. Though the winner of the battle still depends on programmers, the reordering technique provides the key to victory.

1.1 Solitaire, Anyone?

The reordering technique works like an electronic solitaire machine. This machine has a special feature; it allows players to review previous games and make a single change. After the change, players play to the end of the game.

The key to changing either a solitaire game or a parallel program is partial replay. In solitaire, there is almost no chance that two consecutive games will feature the same cards in the same order. Thus, the solitaire machine must control the order of cards for players to reorder past games. The machine works this way: first, it reads moves from a log and replays them. When a player makes a change, the machine checks that the change obeys the rules of solitaire, and then allows the game to continue. In sum, the solitaire machine must repeat past actions to a certain point, and then allow a change.

Similarly, reordering debuggers must also control actions up to a change. But a reordering debugger must control more than a solitaire machine. A solitaire player changes one move. A programmer, on the other hand, changes a whole series of actions. For example, a programmer might reorder a set of actions from {**Gamma**, **Beta**, **Alpha**} to {**Alpha**, **Beta**, **Gamma**}. Like solitaire machines, reordering debuggers must control and check changes. But reordering debuggers perform this duty again and again, for each action in the reordered set.

1.2 A Model: Entities and Actions

We will examine the reordering technique and related work using the *entity model.* An *entity* is an abstract model for a programming language concept like an object or a processor. *Messages* enable entities to communicate with other entities. Each message starts another action. An action is the uninterrupted computation run as the result of one received message. An action can both create new entities and send more messages.

While using the entity model, we will feature three terms: *error*, *location*, and *ordering.* An *error* is a contiguous set of actions that run on one entity. A *location* pinpoints an error. It consists of both an entity and a contiguous set of actions from this entity's execution history. A programmer specifies an ordering of an error location. An *ordering* is the order in which the debugger runs actions from an error. Because the program remains unchanged, the ordering contains the same number of actions as the error.

The entity model applies to many systems, not just message-passing parallel computers. For instance, it applies to both shared-memory parallel computers and clusters of workstations. More generally, it applies to any distributed system that runs uninterrupted actions. Thus, the reordering technique applies broadly. Many programmers will benefit from using it.

1.3 The Reordering Technique

When does a programmer use the reordering technique? When he suspects that an unintended action order has caused an error. Before using this technique, the programmer must hypothesize about what might be causing the error. In the process, he uses other debugging techniques, such as replay debuggers or analysis programs, to gain insight into the error-causing program run.

After examining the program, he has formed hypotheses, orderings that could fix the error. To test his various hypotheses, the programmer uses the reordering technique. Let us consider a specific example involving actions **A**, **B**, and **C**; all three actions run on the same entity. The programmer thinks that the order of these three actions in the log, {**C**, **B**, **A**}, causes an error.

To test this hypothesis, he specifies the ordering {**A**, **B**, **C**}.

The reordering debugger then checks whether, according to its limited information, the new ordering will deadlock. The debugger checks orderings because a programmer who writes incorrect programs is certainly capable of specifying unrunnable orderings. The debugger detects whether an ordering will deadlock based on messages in the log.

Now the debugger runs the program again, controlling it in three stages. First, the debugger must replay enough of the program to reach the reordered actions. This stage ensures that the program will reach the ordering error. Once the program reaches the reordered actions, the debugger enters its second stage, ensuring that all reordered actions occur in the order specified by the programmer. The third stage starts right after the last reordered action. In this final stage, the debugger is forced to run the program uncontrolled. It cannot control the program because variable values might have changed due to the ordering, causing the program to behave differently than in the log.

After the program finishes running, the programmer decides whether the ordering has produced an expected result. If it has not, the programmer has disproved his hypothesis. He then tries another ordering using the log from the original, erroneous program run. Even this result is good since the programmer has eliminated an incorrect hypothesis.

Once the programmer finds an ordering that produces an expected result, he changes the program to produce this ordering. The ordering that he specified serves as a target; unlike when using current debuggers, he knows of one ordering that will produce an expected result.

With this simpler, more comprehensive debugging technique, programmers can fix errors they cannot fix with current debuggers. By ensuring that programs will reach an error when a programmer specifies a new order, the reordering technique unleashes the potential of parallel programmers. Programmers finally have one of the necessary tools to rise to the daunting challenges of parallel programming.

1.4 Related Work

Debugging techniques have one of two agents find an error; either a programmer or a computer program searches for errors. The *replay* and *reordering* techniques provide tools to a programmer. *Static analysis* and *dynamic analysis* approaches force the computer, not the programmer, to find errors.

Programmers Find Errors

Replay is a common technique in current parallel debuggers [LMC87, MC89b, Ber91, NM92]. Replay ensures that parallel programs produce the same output with the same input, enabling *cyclic debugging*. A cycle is one run of a program. Each successive cycle helps the programmer to focus in more closely on an error.

All replay debuggers control programs according to events in a log. Where they differ is in the events they record. Debuggers for programming languages that provide a distributed-memory abstraction [Mal92] record events about messages [Bal69, LR85, Smi84, NM92]. Debuggers for programming languages that provide a shared-memory abstraction record events about reads and writes of shared-memory [LMC87, MC89a, Net93]. As already mentioned, replay debuggers are good for finding an error in a program, but a programmer using a replay debugger can hide an error since he lacks control of action order.

A reordering debugger provides a programmer with a tool to debug programs. The earliest example of a reordering technique appeared in Schiffenbauer's thesis [Sch81]. He built a debugger for a group of distributed Alto computers. Though not his main focus, his debugger let a programmer control message delivery order. His operating environment was a distributed system, though, not a parallel one. He assumed that messages took a long time to travel from one node to another. This characteristic made it easy to control messages. In fact, his interface was interactive because the message transmit time was so high.

The reordering technique is most similar to the design by Goldberg et al. [GGLS91]. They hinted at a reordering technique in their paper about restoring globally consistent states. In their technique, a programmer would reorder concurrent actions on one entity by changing the

order of message receipt. Goldberg et al. did not implement their mechanism. They claimed that the reordering technique is too similar to optimistic recovery, which they had already implemented. But they have delayed showing that the reordering technique is a valuable tool.

Janice Stone has suggested a different type of reordering technique. In this technique, a programmer changes the order of *concurrent events*. Events are not actions since an event can receive more than one message. The main difference between the reordering technique described in this thesis and Stone's technique is not in what they reorder, but how many. In Stone's technique, the programmer focuses on every concurrent event. In small systems, ones with few processors, this approach works fine. Whether a system with many processors could swamp the programmer with events has not yet been proven. But the results in Chapter 5 suggest that this is indeed the case, so the reordering technique presented here appears to be superior.

Computers Find Errors

Both static and dynamic approaches have been used to find ordering errors automatically. Both approaches identify actions that should be mutually exclusive, but can overlap due to missing synchronization. Static approaches try to detect ordering errors in a program without executing it [Tay83, EP89, CKS90]. They find a superset of real ordering errors by considering execution paths that the program never follows. Unfortunately, static approaches require exponential time to find ordering errors.

Dynamic approaches, on the other hand, find ordering errors with a log recorded during a program run. These approaches identify a subset of the existing ordering errors because they analyze just a handful of program executions. There are two types of dynamic approaches for detecting ordering errors: *on-the-fly* [DS90, Sch89] and *post-mortem* [MC89b, AP87]. *On-the-fly* approaches calculate ordering errors at runtime and discard unnecessary data. They produce relatively small logs, but impose a larger overhead at runtime. *Post-mortem* approaches record more data as the program runs and analyze it afterwards.

1.5 Contributions

We have learned that changing the order of actions is not a new idea. But we still face challenges. This thesis will answer these four question:

1. Where should a debugger switch entities to uncontrolled execution?
2. How can the debugger ensure that all reordered actions run?
3. How can the debugger switch to uncontrolled execution?
4. What experiments will best test the utility of the reordering technique?

Questions 1 and 2 address new design challenges in building a reordering debugger. The debugger must first ensure that all reordered actions run. Without this guarantee, the reordering technique is worthless. On the other hand, the debugger must control no actions after the reordered actions. A balance must be struck. There are many actions—*concurrent actions*—that occur neither before nor after any reordered action. The debugger must switch to uncontrolled execution after a concurrent action.

Ideally, a switch would not disturb a program run. Remember that the ordering error happened because of fragile timings among various actions on entities. The fragility of these timings still exists when the debugger changes to uncontrolled mode. If the debugger seriously disturbs the relative timing of actions, the program can behave completely differently after the ordering. This different behavior can delay fixing an error by introducing a new one after the reordered actions.

No debugger can achieve the ideal—not disturbing relative action timings at all—on a real machine.[2] But the debugger must strive for very little disturbance. This is where we answer question 3. Reordering debuggers must avoid centralized solutions, which are conceptually simple, but have high overhead.

[2]It is achievable on a simulator.

Question 4 is crucial. Since Hindsight is the first debugger built for the explicit purpose of reordering program runs, experiments must show its strengths and weaknesses. More importantly, experiments must answer these three questions:

1. Do real programs contain ordering errors?
2. Is there a manageable number of actions in an ordering error?
3. Does a programmer want to focus on one entity?

The answers are: "Yes," "Yes," and "Yes." The results are very positive, suggesting that programmers will want the reordering technique in their arsenal of parallel debugging tools.

1.6 Roadmap

After our introduction, Chapter 2 explains how to use a general reordering technique. Chapter 3 explains the design of Hindsight, which provides a special reordering technique. Chapter 4 describes and evaluates alternative design choices. Chapter 5 puts Hindsight to work, performing experiments to evaluate the reordering technique. The reordering technique tests hypotheses successfully. Chapter 5 also teaches us important lessons about the reordering technique that will help to improve future debuggers. Chapter 6 concludes this thesis and predicts the future of parallel debugging.

Chapter 2

How to Reorder

The reordering technique is easy to use. Debugging just one error is a simple, four-step process. A programmer participates in the first two steps. First, the programmer runs the error-producing program and the debugger records a log of the error. Second, he inputs a new ordering of a set of actions, and some related data. Then the debugger runs the last two steps. It checks to ensure that the new ordering will not deadlock. Fourth, the debugger runs the program, controlling it with the changed log. Controlling a program is like stitching a wound. If a changed order produces an expected result, the programmer modifies the program to produce this order, "healing the wound"; if not, he specifies another ordering.

Debugging a program with more than one error is also simple. A programmer follows the same four-step procedure. There are just two extra requirements: programmers must debug errors that occur one after another last one first; and errors that occur concurrently just require special input data, as described below. Thus, a programmer is equally well equipped to debug multiple errors in one program.

Simplicity is the key feature of using the reordering technique. This technique's four steps, though simple, empower programmers, unleashing their full potential to combat problems that are unique to parallel programs.

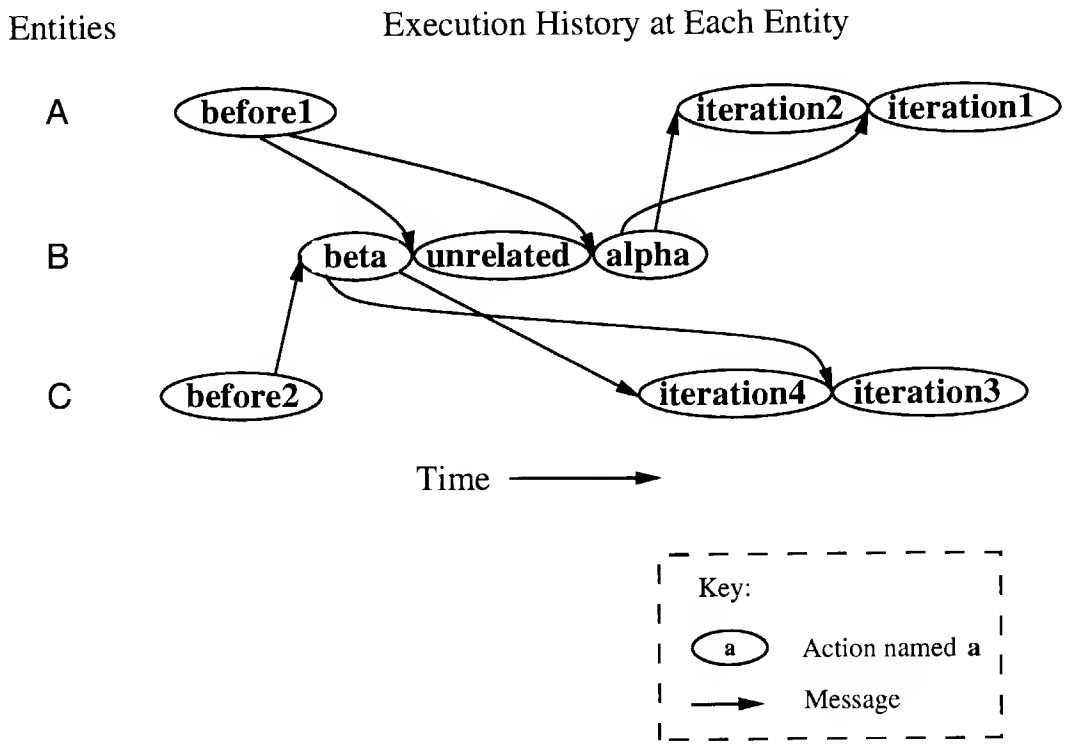


Figure 2-1: A sample execution history

2.1 Recording a Log

The first step to debugging an error is easy: just throw a switch and run the program. The debugger records a log of the order in which actions run on entities. This log will be used in steps two, three, and four.

Figure 2-1 shows the execution history of an example error. In this figure, an ellipse represents an atomic function execution, and an arrow represents a message.[1] The error occurs in entity B, when **beta** runs before **alpha**.

[1] Appendix A explains the shapes and patterns used in figures throughout this thesis.

2.2 Changing the Log

Second, the programmer tells the debugger how to control the program. He inputs two pieces of data: an order to run actions; and instructions on where to switch to uncontrolled execution. The action order is the programmer's stitches for fixing the program, something like {**A**, **B**}. Recall that after this new order, the debugger cannot control the program according to the log order because variable values may have changed. The second piece of data, the switch place, tells the debugger at what time to stop controlling the program. With these two pieces of data, the debugger is almost ready to run the program.

2.2.1 Specifying an Ordering

Specifying an ordering consists of three steps: specifying an entity, a set of actions, and a new order. The first two steps focus a programmer's attention. Once the programmer focuses on a set of actions, he reorders them.

Let us return to our example to follow how to change the order of **alpha** and **beta**. First, the programmer chooses entity B. Second, the programmer chooses a contiguous set of actions from the chosen entity. In this example, he chooses {**beta**, **unrelated**, **alpha**}. Third, the programmer specifies a new order in which the debugger will run chosen actions.

Figure 2-2 illustrates a user interface used to specify a new ordering. At first, the old ordering is displayed on the left and the new ordering is empty. The programmer moves actions from the old ordering to the new one. Once the programmer is satisfied with an ordering, he types "done."[1]

2.2.2 Specifying a Switch Cut

Recall that after the debugger runs an ordering it must stop controlling the program. A *switch cut* tells the debugger where to switch to uncontrolled execution. A switch cut can occur after any actions running concurrently with the ordering. To specify a switch cut, a programmer

[1] If the programmer types done before all of the functions in the old ordering have been included in the new ordering, the debugger tells the programmer to finish specifying a new ordering.

```
Specify a new ordering:

Old ordering:                New Ordering:
1. beta
2. unrelated
3. alpha

choice: 3
```

Figure 2-2: Reordering the chosen set

specifies a *switch point.* A switch point tells the debugger after what action to switch an entity to uncontrolled execution.[2] In our general reordering technique, the programmer specifies a switch point for every non-reordered entity. The switch point for a reordered entity always occurs immediately after the last reordered action.

In this example, the programmer specifies a switch cut comprised of the switch points illustrated in Figure 2-3. The programmer specifies a switch point for entity A from a list of the actions from entity A's execution history. There is also a special case, **BEGINNING**, that means before any actions. Figure 2-4 shows such an interface for this example. To specify the switch point after **before1**, the programmer chooses **before1**. For entity C, the programmer sees a list of choices similar to the one for A in Figure 2-4. The programmer chooses **before2** so that the switch for entity C immediately follows **before2**. The debugger automatically puts the switch point for the reordered entity, B, in the modified ordering right after the last reordered function, **beta**.

This method for specifying a switch cut is simple, almost too simple. When there are many entities, it could be a burden for the programmer to specify a switch point for every entity. Goldberg et al. have presented a less-burdensome method. Instead of specifying a switch point for each entity, the programmer specifies a switch point for just those entities he wants to control exactly. The computer then figures out, if possible, switch points for all remaining entities.

[2]The programmer cannot specify a switch point before an action because the actions that run after a switch point are uncontrolled. Thus, actions different from those in the log can run immediately after a switch point.

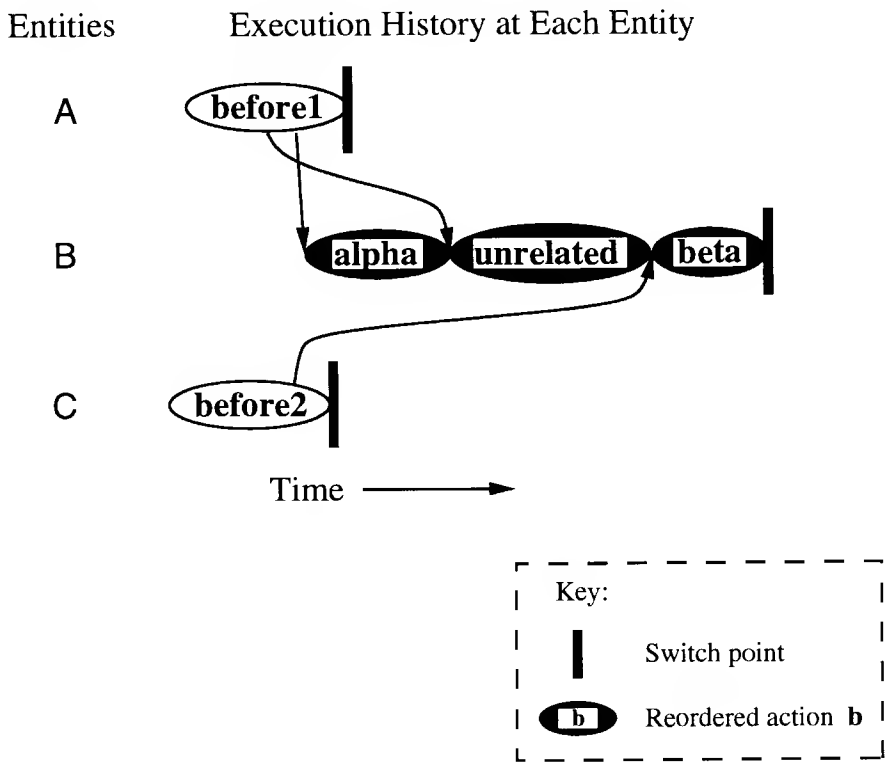


Figure 2-3: The execution history before a switch cut

```
Choose a switch point:

1. BEGINNING
2. before1

choice: 2
```

Figure 2-4: Specifying entity A's switch point

We shall see in the next chapter how Hindsight has reduced programmer burden even further by choosing a particular switch cut for the programmer. This feature distinguishes Hindsight's special reordering technique from this more general one.

2.3 Checking Programmer Input

Third, the debugger checks that the programmer's data obeys certain rules. These rules ensure that no controlled action depends on an uncontrolled action. The reason is simple. The uncontrolled action can run differently; specifically, it can send different messages, resulting in the controlled action not running. Meanwhile, the debugger deadlocks while waiting for the controlled action to start.

2.3.1 Checking the Ordering

The debugger uses logged messages to check the ordering. It looks for a path of messages from one reordered action to another. Any such path is an error. The reason is that actions after the first reordered action are uncontrolled. Thus, the second reordered action, at the end of the path, depends on uncontrolled actions. As mentioned above, this situation can cause deadlock, so when the debugger finds such a path it signals an error. Then it asks the programmer for another set of actions to reorder.

Let us see why a reordered action can run with different variable values. Consider the set with the specified ordering, {**A**, **B**, **C**, **D**}. The programmer places **A** first since it is the first action to run. **A** changes the variables used by the following actions. The first action in an ordering will generally differ from the one in the original order. If it were the same, there would be no reason to include it in the ordering. When the first action is different from the log, later actions can use different variable values and are liable to start different actions. Thus, there cannot be a path of messages from one reordered action to another; the first reordered action may not send the message to start this path.

In sum, programmers must not reorder actions connected by a path of messages. This requirement is easily met, especially in comparison with specifying a switch cut, which must

obey more complex rules.

2.3.2 Checking Switch Cuts

Three rules govern switch cuts and avoid deadlock:

1. In each entity, there is a set of actions that happen before any reordered action. The switch point for the entity must occur after these actions.

2. Likewise, each entity has a set of actions that happen after any reordered action. The entity's switch point must occur before this set of actions.

3. Now consider a global property among entities. Choose an action, **B**, before any entity's switch point. Then choose another action, **A**, that occurs before **B**, whether on the same entity or not. Action **B** must occur before the switch point for its entity.

Rule 1 ensures that a reordering debugger can replay enough of the execution to reach every reordered action. If the switch points had occurred earlier, all reordered actions may not have been invoked; the reordering debugger would deadlock while waiting for the reordered actions to start.

Figure 2-5 shows an example that violates this rule. The programmer has specified the switch points of entities D2 and D3 to be too early. In this figure, actions **A** and **C** are uncontrolled. Figure 2-6 illustrates a possible result of this lacking control: actions **A** and **B** on D2 run in a different order. As a result, **C** does not call one of the reordered actions, **E**.

Rule 2 ensures that the switch points do not constrain too much of the execution. Specifically, the debugger must not control action order after the reordered actions. The reason is clear: reordered actions may change variable values, so the program may run in a different order from the log. If the debugger tries to control the program with a log that has a different order, the debugger will deadlock.

Rule 3 guarantees that the debugger controls all actions that happen before a switch cut. Like rules 1 and 2, deadlock is the consequence of violating rule 3. The debugger would not

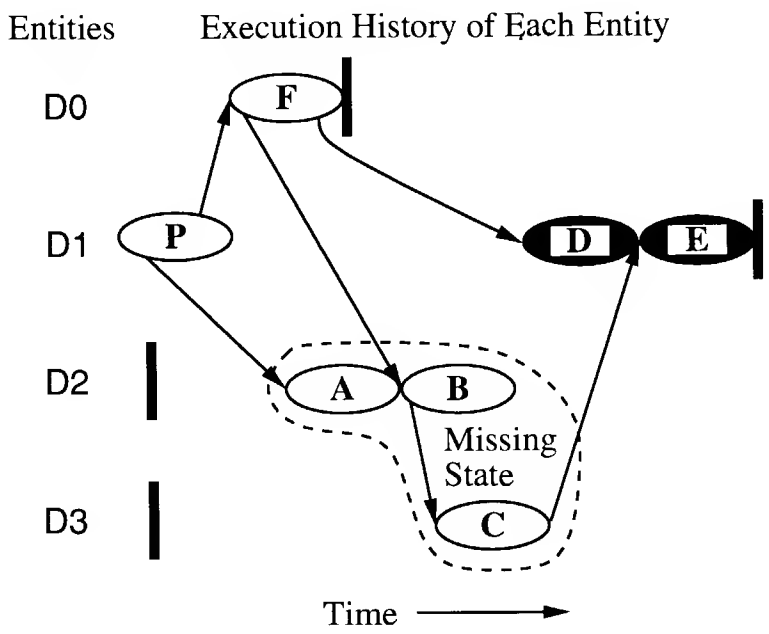


Figure 2-5: Missing state

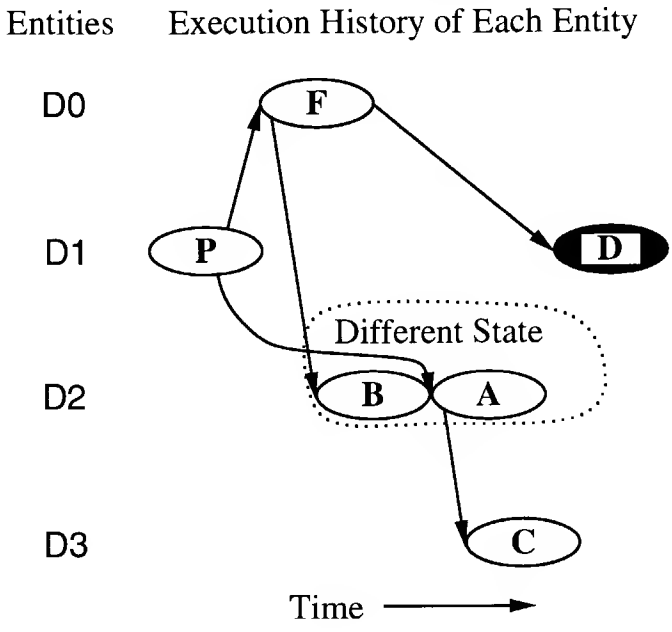


Figure 2-6: Incorrect state caused by missing state

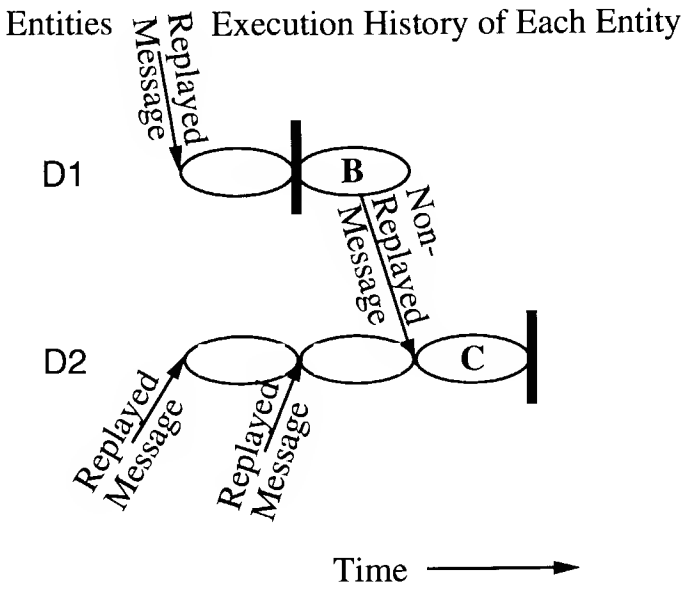


Figure 2-7: Early switch point

control enough actions, so the reordered actions might not start, causing the debugger to deadlock, waiting for the reordered actions to run.

A programmer violates rule 3 when he specifies a switch point that is too early, as shown in Figure 2-7. In this figure, D1's switch point is too early. It should be after **B**. **B** happens before **C**, which happens before D2's switch point.

These rules burden a programmer. Thus, having Hindsight choose switch cuts aids the programmer. Hindsight has shifted one computer duty, checking a switch cut, to another duty, finding a switch cut. As duties go, finding a switch cut is a small one; more important is running a modified order.

2.4 Running in a Modified Order

Fourth, the debugger turns on logging and starts a new program execution. The debugger makes the program execution match the order in the log until the switch cut. As mentioned, after that the debugger lets the program run without control. Once the program finishes, the programmer examines the output. When the program produces an expected result, the

programmer can either change the program to guarantee the specified ordering or test other hypotheses with the log recorded during the controlled execution. When the program produces an unexpected result, the programmer discards the log from the controlled execution and tests different hypotheses.

We have now seen the four steps to reordering one error. From recording a log to watching the debugger run a modified execution, the programmer enjoys a tool with a simple interface, yet lots of power. The reordering technique's power and simplicity are not diminished when programmers face the more daunting task of debugging more than one error in the same program.

2.5 Multiple Ordering Errors

Now we will learn how simple it is to debug multiple ordering errors in a single program execution. There are two types of multiple ordering errors: concurrent and sequential. Each type describes the relation between two ordering errors. *Concurrent errors* occur at the same time. *Sequential errors* occur one after another.

Let us return to the example used in the one-error explanation. Now assume that the program shown in Figure 2-1 has three ordering errors: **beta** incorrectly precedes **alpha** on entity B; **iteration 1** should precede **iteration 2** on entity A; and **iteration 3** should precede **iteration 4** on entity C. The errors caused by **iteration 2** and **iteration 1** as well as **iteration 4** and **iteration 3** occur concurrently. Both these errors are sequential with the error caused by **beta** and **alpha**.

2.5.1 Concurrent Errors

To handle concurrent errors, the programmer must specify a special switch cut. This switch cut has switch points after all errors concurrent with the reordered one. While the programmer debugs one error, this switch cut effectively puts the rest of the errors on ice. Once the programmer has debugged this error, he can debug the others.

In this example, the programmer reorders {**iteration 4**, **iteration 3**} to {**iteration 3**, **it-**

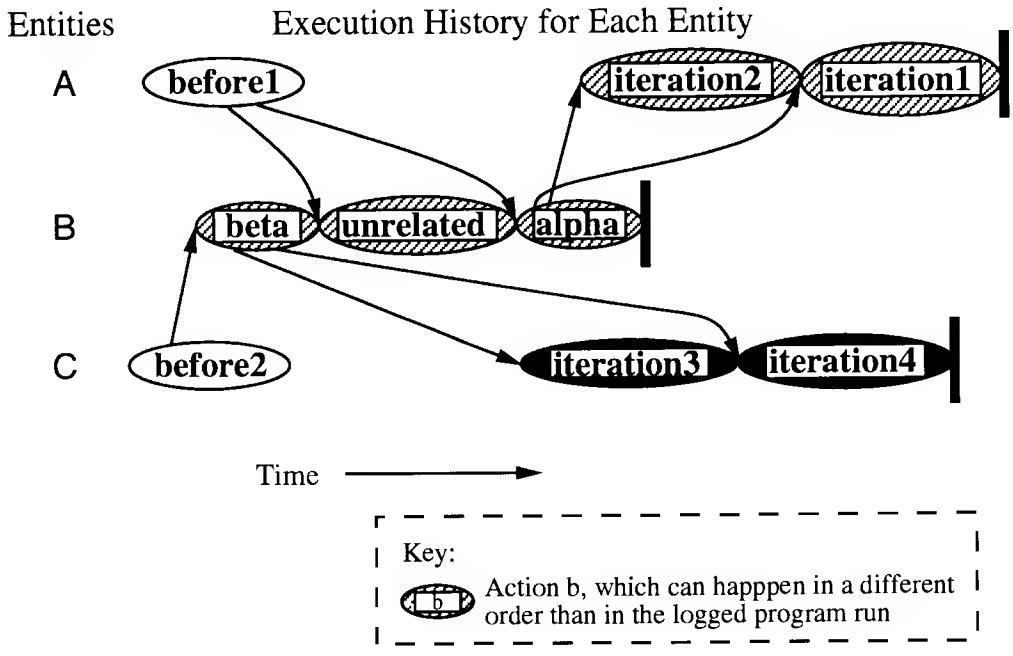


Figure 2-8: Switch cut for reordering **iteration 3** and **iteration 4**

eration 4}. He then specifies the switch cut as shown in Figure 2-8. This switch cut happens after the other error, {**iteration 2, iteration 1**}.

Now we have learned how simple it is to debug concurrent errors. Debugging sequential errors is even simpler.

2.5.2 Sequential Errors

Sequential errors require no special data to be fixed. All the programmer must do is attack them in the correct order: last error first. The logic is similar to the concurrent error case. The programmer wants to put the rest of the errors on ice while working on one error. Recall that the debugger replays all actions before the ordering. Thus by working on the last one first, the debugger replays earlier errors.

In the example with three errors, the programmer reorders a concurrent error first. The reason is now clear: the concurrent errors happen after the {**alpha, beta**} error. After fixing both concurrent errors, the programmer can debug the {**alpha, beta**} error.

2.6 Summary

In this chapter, we have learned how to use the reordering technique. We have seen the reordering technique handle programs with one or multiple errors. Debugging one error is a four-step process:

- A programmer runs a program and the debugger records a log.
- The programmer inputs an ordering and a switch cut.
- The debugger checks the ordering and switch cut for deadlock conditions.
- The debugger starts the program again and controls its execution before the ordering.

To reorder more than one error, a programmer debugs one error at a time, using the same four-step process. For concurrent errors, a programmer must specify the switch cut after all concurrent errors. For sequential errors, the programmer must reorder the last error first.

The reordering technique provides great benefits to parallel programmers. In the next chapter, we will learn how easy it is to implement.

Chapter 3

Prototype

Recall the solitaire machine from the introduction. It allows a player to instruct the machine to read events from a log and re-run an old game until some midpoint. At this midpoint, the player plays one card differently from the original game. The machine checks that this new play follows solitaire rules. If it does, the machine switches from re-running to normal game play. Accordingly, such a solitaire machine needs components to carry out five tasks: logging, replaying, specifying a different move, checking, and switching.

As with a solitaire machine, so with a reordering debugger. Like the solitaire machine's log of moves, the reordering debugger's *logging component* records all actions occurring on every entity. The log plays a central role in the solitaire machine; it is even more important in the reordering debugger. All components except the switching component use information from the log. The *replay component* re-runs actions in the order defined by the log. The *specification component* lets the programmer specify action order, according to names derived from the log. The *checking component* tests whether the ordering will deadlock.[1] The checking component uses the log, not the program, to check for deadlock. The *switching component* changes what parts of the runtime system the debugger controls since the debugger must control different parts during replay, reordering, and uncontrolled execution.

[1]It cannot always tell if the program will deadlock, as that would solve the halting problem, but it can detect many deadlocks that would result from a specified ordering.

If these components were all that reordering debuggers require, they would be almost identical to solitaire machines. In addition, a reordering debugger requires a reordering component. A solitaire machine, with an initial change of just one move, does not need a reordering component. But in the reordering technique, a programmer initially changes a set of actions. Thus, a reordering debugger needs a component to control this set. The *reordering component* imposes the programmer-specified ordering on these actions.

Figure 3-1 illustrates the six interdependent components of a reordering debugger. The components are arranged into four columns, one column for each step in the reordering technique. The interdependencies are illustrated with arrows. Information follows the arrows from one component to another.

Before learning the details of our prototype, Hindsight, we must understand the context in which it has been implemented. Hindsight works with Concurrent Scheme, which runs on the Mayfly processor [Dav92].

3.1 Context: Concurrent Scheme

Concurrent Scheme is a language formed by adding parallel constructs to the sequential language Scheme. The entities in Concurrent Scheme are domains, ports, and placeholders. Domains are the most general entities because they can run any user-defined action.

Ports and placeholders are specialized entities. A port is a first-in-first-out (FIFO) queue; it runs only the system-defined *enqueue* and *dequeue* actions. A placeholder is a write-once variable. It can run the system-defined constructs *determine* and *touch*. *Determine* writes the variable; *touch* reads it. Hindsight handles programs with placeholders, but not ones with ports.

Concurrent Scheme augments Scheme with the *make-thread*, *apply-within-domain*, *delegate*, and *emigrate* constructs. *Make-thread* starts actions asynchronously; the caller continues local computation without waiting for a result. *Apply-within-domain* is a remote procedure call (RPC) that blocks the sending action [BN84]. The caller waits for a response without letting another action run. *Delegate* is a non-blocking RPC; it also waits for a result, but allows other actions to run before the response arrives. *Emigrate* does not wait for a result. Instead, it sends

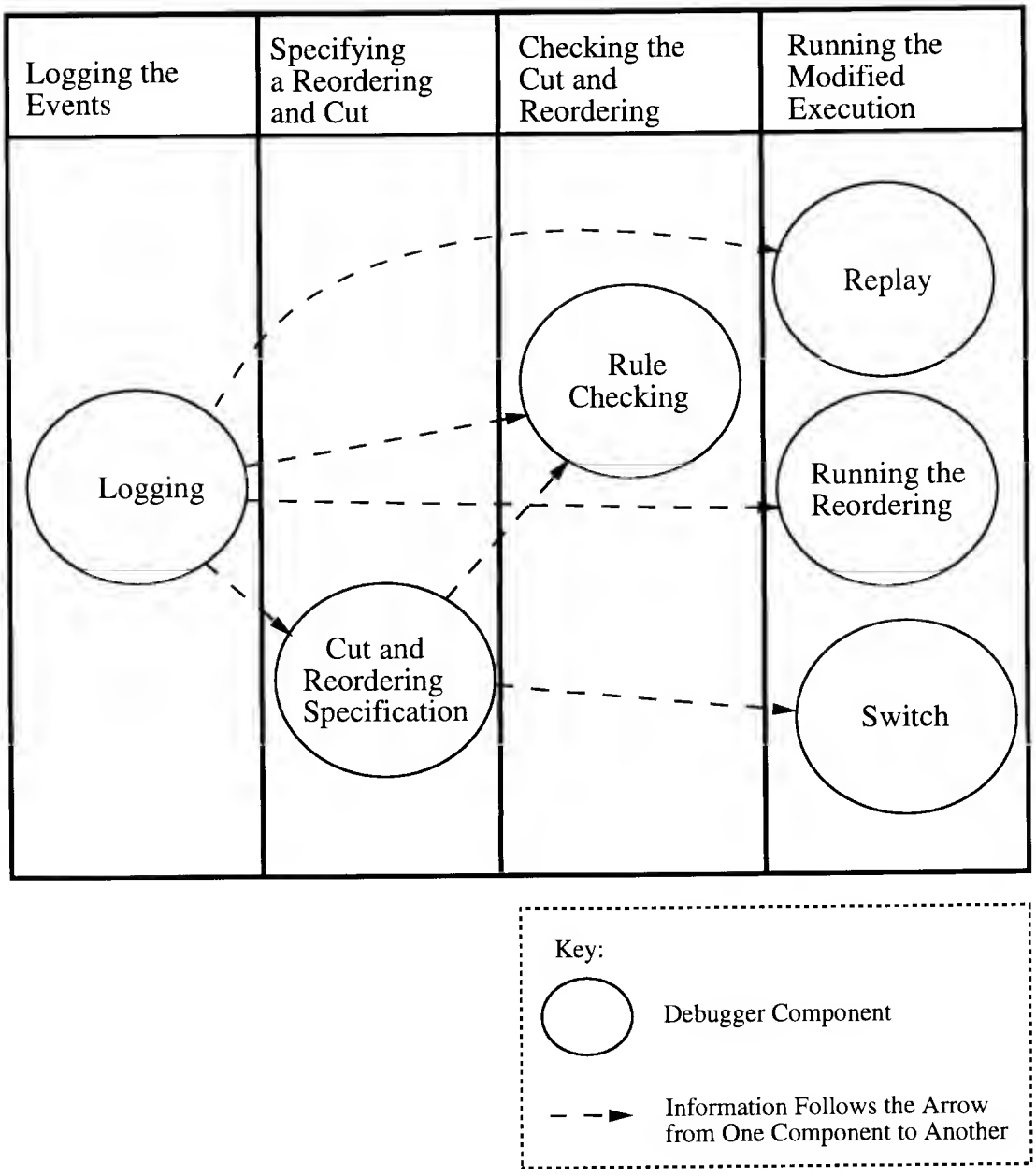


Figure 3-1: A reordering debugger's components

the rest of the computation along with the message. The message recipient then continues the local computation.

Figure 3-2 illustrates Concurrent Scheme's communication constructs. The make-thread picture shows three calls from **A**. These calls show that an action like **A** can send multiple messages asynchronously.

The other communication constructs end an action by sending a message. Apply-within-domain's reply message starts a new action. Delegate's reply also starts a new action. The other actions that occur between a delegate's send and reply are illustrated by the black circles in Figure 3-2. Emigrate's send message ends an action since the caller's action continues on the recipient's entity.

Now that we understand the context in which Hindsight has been built, let us examine its components.

3.2 Recording Events in a Log

Hindsight records events in a log. It records events about each message send, message receive, action, and entity creation. Each processor records events into a *processor log*. A processor log holds data about every entity on the processor. The debugger intersperses events concerning different entities within the processor log.

Figure 3-3 depicts the fields in Hindsight events. All events of the same type have constant size. Even the action execution and entity creation events, which include names, have constant size. Hindsight accomplishes this feat by recording a number instead of a string name. Hindsight obtains string names from a table that maps from numbers to string names; the Concurrent Scheme compiler writes this table.

There are two advantages of different-sized event types. First, Hindsight can record smaller logs; the smaller event types need not record empty space to be the same size as the larger event types. Second, Hindsight has a more modular design; programmers can change the size of one event without affecting the size of other events.

Some types of fields are common among various event types. Hindsight uses these common

```
MESSAGE SEND:
        event identifier
        sending entity identifier
        target node
        message identifier

MESSAGE RECEIVE:
        event identifier
        receiving entity
        sending node
        message identifier

ACTION EXECUTION:
        event identifier
        entity that runs action
        message identifier
        action name

ENTITY CREATION:
        event identifier
        creating entity identifier
        created entity identifier
        entity name
```

Figure 3-3: Hindsight's log events

fields for basic functions like traversing and examining the log. For instance, every event has an `event identifier` field. Hindsight uses it for two purposes: to advance to the next event; and to examine the fields of the current event.

The other fields in each event type differentiate each event from other events of the same type. For example, a message receive can be uniquely identified by its `sending node`, `message identifier`, and `receiving entity` fields. Likewise, an action execution event can be distinguished from other action execution events by its `action name` and `message identifier` fields.

Let us step back and put this logging component in perspective. First, this set of events may seem repetitive. For instance, if Hindsight records message receives, why does it also record message sends? The answer is that Concurrent Scheme allocates identifiers in a timing-

dependent manner. It allocates identifiers per processor, though multiple entities on a processor can run in any order.[2] Since Hindsight controls action order with identifiers, they must be the same in the logged and replayed program runs. Thus, Hindsight must record the otherwise extraneous message send and entity creation events.

More generally, Hindsight's logging component embodies the most efficient logging techniques known when it was designed.[3] It avoids the earlier pitfall of recording arguments to actions, recording just the order of actions instead.

Midway through our development of Hindsight, Professor Robert Netzer of Brown University published a vastly improved logging technique. His technique selectively decides which events to record. Hindsight does not include this improvement; to do so would change many of the other components. But Hindsight is ultimately limited by the size of the logs it records; it cannot handle very long running programs. Because Netzer's logging technique is important, the design alternatives chapter will show how a reordering debugger can use that technique.

Regardless of Hindsight's logging technique, its other components must use data which it records. The first such component that the programmer comes in contact with is the specification component. He uses this component to specify a different ordering.

3.3 How to Specify a Different Ordering

In Hindsight, the debugger and programmer each specify part of an ordering. The programmer reorders a contiguous set of actions on one entity, and the debugger provides a switch cut. The programmer can reorder actions that run on domains only, not those running on ports or placeholders. Hindsight does not control entities after the *earliest cut*; the programmer does not choose a switch cut.

The earliest cut is a quasi-switch-cut since it does not necessarily include a switch point for every entity. An included entity's switch point occurs after the last action that happens before any reordered action. The excluded entities contain no action that happens before a reordered

[2] We will learn in the design alternatives chapter why Hindsight cannot replay per processor.
[3] 1991

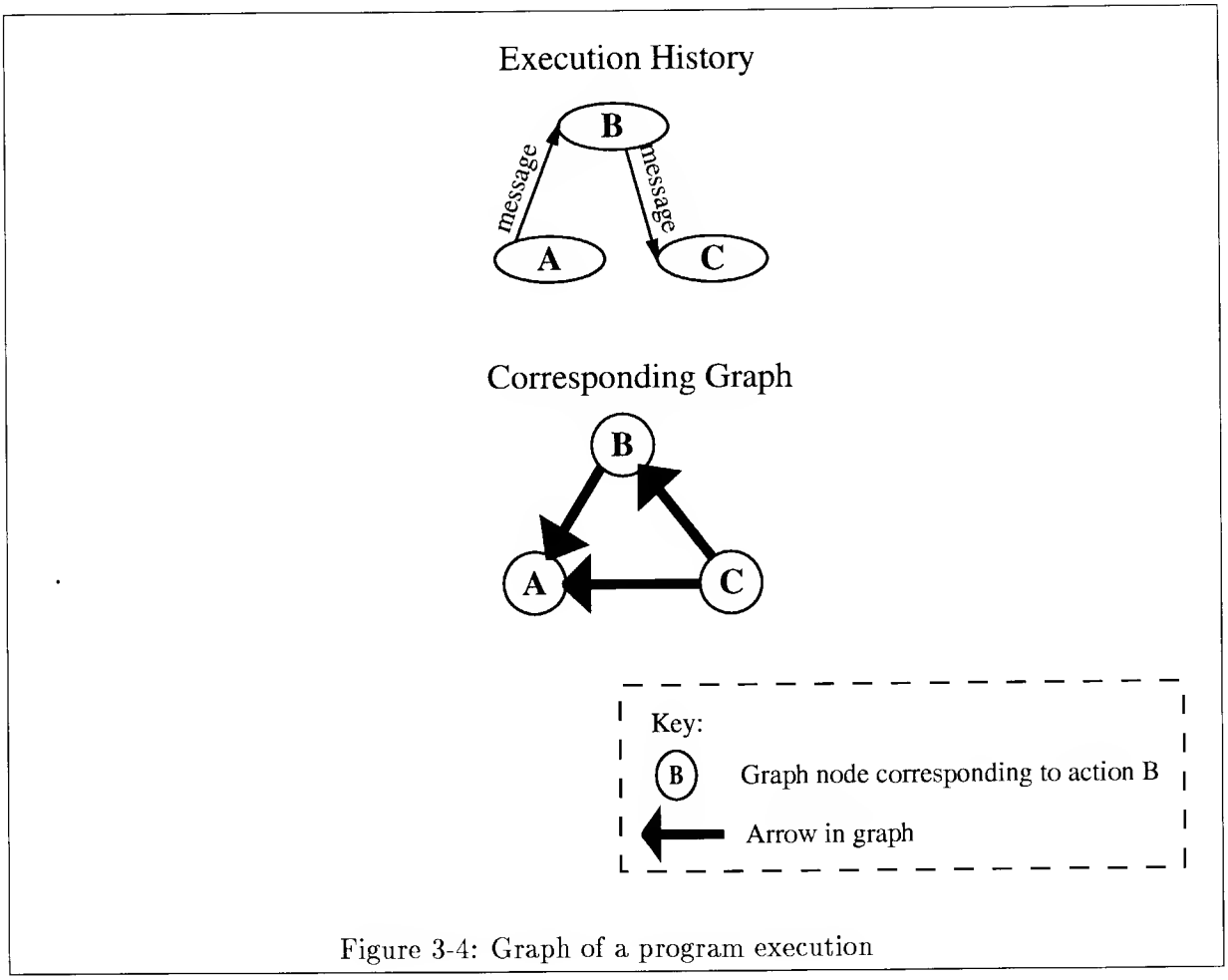


Figure 3-4: Graph of a program execution

action. Thus, Hindsight need not control such an entity to reach a reordered action.

Hindsight calculates the earliest cut with a graph traversal algorithm [Law76]. The graph allows Hindsight to find which actions occurred before other actions. A graph of a program execution is a directed, acyclic graph. A node in the graph represents an action; an edge represents either a message or a time dependency on the same processor. An edge representing a message starts at a receiver and ends at the sender because the algorithm determines which actions happened before others. Similarly, a time dependency edge stretches from a later action to the previous one.

Figure 3-4 illustrates an example execution history and its corresponding graph representation. During this program execution, action **A** sent a message to action **B**, which in turn sent a message to action **C**. Thus, in the corresponding graph representation, there is an arrow from

node **C** to node **B** and from node **B** to node **A**. A time-dependence arrow stretches from node **C** to **A** since action **C** happened after **A** on the same entity.

In summary, the specification component is responsible for two key choices in the reordering process: the programmer's choice of ordering and the debugger's choice of switch cut. These two choices provide the data needed to run a different order and to switch entities from controlled to uncontrolled mode at the correct time. But the debugger is programmed not to trust the programmer fully; it checks the programmer's ordering choice.

3.4 Checking an Ordering

Recall from Section 2.3 that Hindsight checks the programmer's choice of ordering for deadlock. It does so by looking for message paths from one reordered action to another. A *message path* is a chain of messages connecting one reordered action with another reordered action. Since the debugger cannot control any action after a reordered action, the messages in the middle of a message path—the ones after the first reordered action—are uncontrolled. A message path causes trouble since the reordered action at the end of the path depends on messages sent by uncontrolled actions. The debugger cannot depend on uncontrolled actions without risking deadlock. The program would deadlock when the uncontrolled actions run differently.

Hindsight must conceptually check all of the paths starting at a reordered action. This conceptual check, however, requires lots of time. Ideally, Hindsight could check an ordering in no time. Indeed, it can.

To do so, we must view the problem backwards. That is, any path that leads from action **A** to action **B** must also lead backwards from action **B** to action **A**. Hindsight already has a function that searches through backward paths. This function finds the earliest switch cut. To add checking of orderings to this function, Hindsight just detects reordered actions while this function searches paths.

To combine these operations, Hindsight sets the switch point for the reordered domain immediately before the first reordered action. Normally, this switch point occurs after the last reordered action, where it is never changed during the computation. By placing this switch

point before all reordered actions, the debugger will try to update this switch point when it finds a later action. The only possible later action is a reordered one since the debugger searches backwards, starting at the reordered actions. Thus, when Hindsight tries to update the reordered domain's switch point, it asks the programmer for a different ordering.

Hindsight does not have to check the earliest cut since it always obeys the switch cut rules. To review, these rules are:

1. In each entity, there is a set of actions that happen *before* any reordered action. The switch point for the entity must occur *after* these actions.

2. Likewise, each entity has a set of actions that happen *after* any reordered action. The entity's switch point must occur *before* this set of actions.

3. Now consider a global property among entities. For any entity, choose an action, **A**, occurring before its switch point. No action that happens before **A**—whether on the same entity or not—can occur after the switch point for its entity.

The earliest cut obeys rule 1, ensuring that the switch cut does not happen too early. The earliest cut also satisfies rule 2, ensuring that a cut does not occur too late. The earliest cut obeys rule 3; indeed, because the earliest cut is found by transitive application of the third rule.

In conclusion, having the debugger supply the earliest cut simplifies the debugger's task as well as the programmer's. For reasons we have just followed, the earliest cut always obeys switch cut rules. Thus, the debugger need not check the earliest cut, saving it time. The programmer is also spared the job of understanding all concurrent actions to specify a switch cut.

We have learned how Hindsight performs preliminary duties: finding the earliest cut and checking the ordering. Now let us examine the core of a reordering debugger, the replay component.

3.5 Replay: Before the Switch Cut

Hindsight's replay component reads events from the log and controls all action executions, message sends and entity creations. We will first learn how Hindsight carries out the crucial task of controlling actions. Then we will learn how Hindsight performs supporting duties: controlling message sends and entity creations. All told, these duties ensure that all actions in the ordering will run with the same variable values as in the logged execution.

3.5.1 How Hindsight Controls Actions

Hindsight controls actions at two different times: when messages arrive and when actions leave. When a message arrives, Hindsight checks whether this message carries the next action in its target entity. Hindsight performs this check by comparing the message identifier with the next identifier in the log. When searching for the next identifier, Hindsight considers just messages arriving at the target entity. If the identifiers match, Hindsight checks if the target entity is running an action. If so, Hindsight enqueues the message; if not, the target entity runs the action which the message brought.

Once an action finishes, Hindsight checks the message queue for the next action to run in the entity. If the next one is found, Hindsight dequeues and runs it. If not, the entity sits idle, waiting for the next message to arrive. Figure 3-5 shows these rules in pseudocode.

These simple rules controlling action order may seem efficient, but searching queues can take time. To overcome this problem, Hindsight employs *system queues.*

Storing Messages in a System Queue

An action can enter a domain through either the general queue or a delay queue. These multiple entry points would slow down replay, if Hindsight did not store messages in a system queue. As mentioned, when an action finishes, the debugger must look among all received messages for the next action to be run. To avoid scanning empty queues, Hindsight stores messages in a system queue. One system queue is associated with each controlled domain. Hindsight puts messages arriving too early at the tail of the system queue, and thereby preserves the order in

when a message arrives at an entity:

```
if this message identifier matches the next
  one to enter the entity according to the log

    then if no other action is executing

        then start executing on the entity.
        else enqueue on the entity's queue.

    else enqueue on the entity's queue.
```

when an action leaves an entity:

```
if a message in the entity's queue should start the next
  action

    then run the action brought by the message with the
      matching identifier
```

Figure 3-5: How Hindsight controls action order

which messages are received. If messages arrive in an order close to that in which they will run, Hindsight finds them quickly.

System queues reduce the time needed to locate the next action. They do this by consolidating all of an entity's waiting messages in one queue. We have seen how Hindsight controls action order efficiently. Now let us see how Hindsight carries out two supporting duties: controlling entity creations and message sends.

3.5.2 Controlling Entity and Message Identifiers

If Hindsight controls action executions, why must it also control message sends and entity creations? The reason: to control message and entity identifiers. Concurrent Scheme's runtime system allocates identifiers in a timing-dependent manner. The result is identifiers that do not match those in a log. As we just saw, though, Hindsight needs identifiers to match those in the log in order to control action order. To do this, Hindsight must control entity creations and message sends.

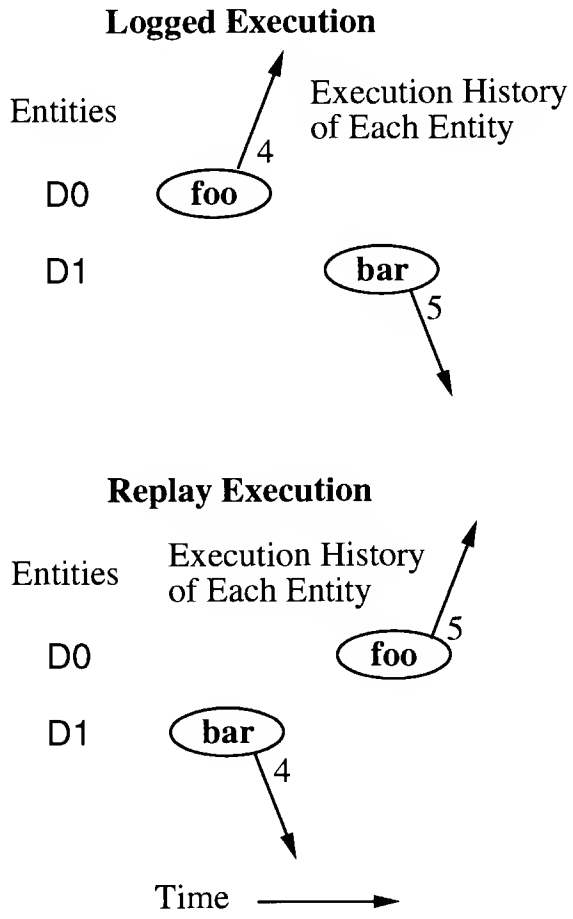


Figure 3-6: Messages are assigned different identifiers during replay and uncontrolled execution.

To ensure matching identifiers, Hindsight *patches* identifiers during replay. To patch an identifier, Hindsight locates the next identifier in the log and assigns it to a new message or entity. The patched identifiers ensure that Hindsight can control action execution according to the entity and message identifiers in the log.

Hindsight controls message and entity identifiers in the same way. The following discussion explains the methods and potential problems of controlling message identifiers. The same ideas apply to controlling entity identifiers.

Consider the example shown in Figure 3-6. Figure 3-6 shows non-matching message identifiers that could result if Hindsight did not control message identifiers. When Hindsight replays

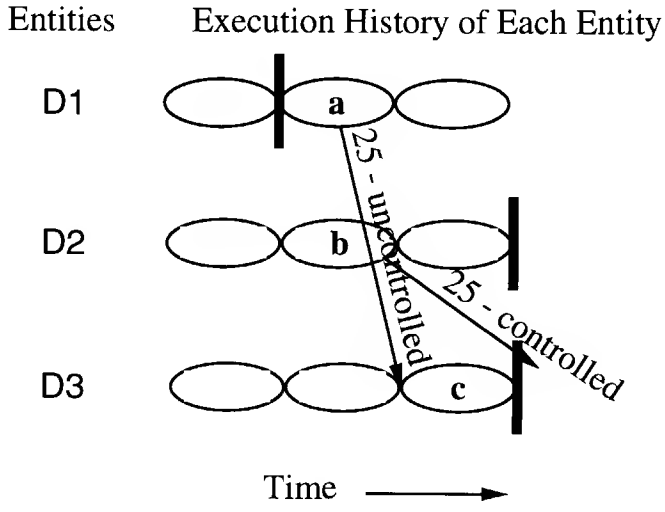


Figure 3-7: Duplicate message identifiers

this execution, it assigns message identifiers from the log. When, at the bottom of the figure, **bar** sends a message, Hindsight searches the log for the next message sent by entity D1, and finds message identifier 5. Hindsight then patches the message's identifier to 5—not 4, as the system would have done. Similarly, when **foo** sends a message, Hindsight patches the message's identifier to 4, not to 5.

Interestingly, sometimes a message identifier from a controlled entity can match a message identifier from an uncontrolled entity. When this occurs, the uncontrolled runtime system assigns the same identifier as found in the log. Figure 3-7 shows an execution history with a duplicate message identifier. D3 could accept the uncontrolled message from D1, when it should accept the controlled message from D2. D1 sends a message to D3 after its switch point. The message has identifier 25 since D1 is not controlled. The runtime system assigns the next message identifier, which happens to be 25. At almost the same time, D2, which the debugger still controls, sends a message to D3. The debugger patches the message's identifier to be 25 in order to match the one in the log.

To eliminate duplicate message identifiers, Hindsight makes all uncontrolled identifiers larger than any controlled one. To do this, Hindsight must find the maximum identifier sent before the switch cut, then broadcast this maximum identifier to all entities before replay begins. This

simple fix is easy to implement, yet effective.

Identifier control is an essential part of the replay component; it allows control of action execution, which is the crucial part of replay. Hindsight's identifier control is not the most obvious solution. The Concurrent Scheme runtime system allocates identifiers per processor, so it may seem obvious to replay per processor, not per entity. But replaying per processor fails. The switch points for different entities can occur at different points in the log. To reach later entities, Hindsight would have to control past some switch points. The design alternatives chapter will further explore this replay pitfall.

3.5.3 The Replay Component: A Summary

The replay component is the workhorse of a reordering debugger. It ensures that all reordered actions will run with the same variable values. We have learned that the essential part of replay is the control of action executions. Hindsight also controls message and entity identifiers to ensure that they match identifiers in the log. Now let us learn about a close relative of the replay component, the reordering component.

3.6 Running an Ordering

While running an ordering, Hindsight controls action order, just as in replay. The difference is that when controlling reordered actions, Hindsight does not patch message or entity identifiers. It must not do so since the reordered actions will likely run with different variable values. Since a reordered action can run differently, the program can send different messages and create different entities. If Hindsight were to try to control these identifiers, it would deadlock when the program sent different messages or created different entities.

The reordering component, a cruder version of the replay component, fulfills a crucial, unsung role. We have now seen how Hindsight controls a program in one of three ways: replayed, reordered, and uncontrolled execution. We will now learn how Hindsight switches among these types of control.

3.7 Switching among Control Modes

There are three periods in an entity's controlled execution: before the switch cut; during the reordering; and after the switch cut. Each period corresponds to an entity mode: replay mode, reordering mode, and uncontrolled mode. Hindsight controls different parts of an entity's execution in each mode. The switching component must change an entity's mode so that Hindsight controls the right events at the right time.

Hindsight independently switches entities among modes; that is, each entity can switch without the permission of any other entity. Independent switching is good because it perturbs the relative timing of entities less than coordinated switching.

The only entity to enter reordering mode is the one for which the programmer specified a different action order. It is the only entity to switch from replay to reordering mode. This change merely throws a switch; the debugger changes no data structures.

The transition to uncontrolled mode is almost as simple. Entities switch to uncontrolled mode from either replay or reordering mode. The entity with a different action order switches from reordering to uncontrolled mode; all other entities switch from replay to uncontrolled mode. At the end of each action, Hindsight checks whether the time to switch has arrived. Once it has arrived, Hindsight changes the system queue from the one used to control this entity during replay to the queues used by the Concurrent Scheme runtime system.[4]

Independent switching is crucial to the reordering technique because it perturbs a program execution so little. The programmer wants a program to run as if the debugger had never controlled it. The only way to ensure this is for debugger control not to disturb the execution timing. The debugger cannot eliminate this perturbation, but it must minimize it. Hindsight does this well by using the independent switching method, which avoids delaying some entities longer than others. In the next chapter, we will learn about a more obvious but inferior solution, the barrier-switching method.

[4]There is one interesting detail in transferring queues. Hindsight preserves message receipt order while moving elements from the system queue to the Concurrent Scheme queues. This preserves the network ordering of messages. Without this feature, Hindsight would exhibit bad design; it would change the features of the many networks that preserve message order.

Chapter 4

Design Alternatives

We now know that some of Hindsight's components are well-designed, like its independent mode switches. Hindsight has wisely avoided five obvious, but inferior solutions; we will call these solutions pitfalls. We explain them so others will avoid them in designing future reordering debuggers.

We also know that Hindsight can still use further refinement. For example, Hindsight ought to record far smaller logs. In this chapter, we suggest two refinements that would make a reordering debugger more efficient and easier to use. Programmers building a new reordering debugger should include both of these refinements.

4.1 Hindsight Design: Avoiding Five Main Pitfalls

Hindsight wisely avoids five design pitfalls. First, Hindsight shuns switching all entities at once to uncontrolled execution. Second, Hindsight avoids requiring the user to reorder all concurrent actions. Third, Hindsight does not allow a programmer to add actions to an ordering. Fourth, Hindsight does not ensure that a set of actions occurs in a certain order throughout an entire program run. And fifth, Hindsight shuns replaying per processor.

These pitfalls have a range of consequences, from efficiency to correctness. The first pitfall, switching entities at once, would decrease debugger efficiency. The second pitfall, reordering all concurrent actions, would decrease programmer efficiency. The third pitfall, adding actions,

would prevent Hindsight from checking an ordering. The fourth pitfall, always ensuring a certain action order, would cause a reordering debugger to deadlock frequently. The fifth pitfall, replaying per processor, affects correctness; the reordering debugger would control actions it promised not to.

4.1.1 Switching All Entities at Once

Recall that Hindsight controls a program execution in three modes: replay, reordering, and uncontrolled. The debugger independently changes the entities that embody the program from one mode to the next; that is, the debugger does not wait for other entities before switching any particular entity. Thus, modes are mixed: the debugger controls some entities, but others run uncontrolled.

An alternative to Hindsight's independent switching method is the *barrier-switch method*, in which each entity reaches its switch point and stops. The entity starts again when all entities have reached their switch points. No entity starts uncontrolled execution until all entities have finished controlled execution. There is no mode mix.

Hindsight's independent switching method is better than the barrier method in three ways. First, it requires no communication with the barrier. Second, it requires less buffer space. Entities need not enqueue messages while waiting for other entities to reach their switch points. Third, the independent switching method affects the relative timing of entities less than the barrier-switch method.

The barrier method delays entities more if they reach the barrier early than if they arrive late. Reordering debuggers should clearly switch entities independently. This improves debugger efficiency while controlling the program. Likewise, the debugger can help improve programmer efficiency by providing the means for programmers to reorder a small set of actions.

4.1.2 Reordering All Concurrent Actions

There are two ways to specify a reordering: either reorder all concurrent actions or choose which actions to reorder. Hindsight wisely lets programmers choose actions to reorder. Requiring programmers to reorder all concurrent actions is one of the five pitfalls; when a program runs many concurrent actions, programmers can become swamped with data.

With only the concurrent actions **A** through **Z**, we can see why it is wise for Hindsight to let programmers choose actions to reorder. Reordering all 26 concurrent actions takes more time than locating the handful of actions causing the error. For programs with more concurrent actions, reordering them all makes the debugger bewildering and time-consuming to use.

These first two pitfalls have affected only efficiency. The others are more serious, demanding much more attention from future programmers. The next pitfall, adding actions to an ordering, fundamentally impairs a debugger's ability to check an ordering. Without checking, reordered programs can frequently deadlock.

4.1.3 Adding Actions to an Ordering

The reordering technique lets programmers change action order. This feature suffices for running an unchanged program in a different order. But programmers must ensure that a changed program runs in a proper order. Programmers thus need new debugger features that let them add and delete actions from an ordering. As we will see, adding actions is not a good idea because the debugger cannot check whether the program will deadlock. But deleting actions is a useful feature; we will learn in Section 5.1.1 how to delete actions by reordering non-contiguous sets.

Adding actions to an ordering appears to be a useful debugger feature. Indeed, one common way programmers add needed control to a program is to add an action. Programmers change the program so that it starts an action synchronously instead of asynchronously. The synchronous action sends a result message to the calling action, where the asynchronous action does not. The result message starts an action that did not run in the logged execution. By adding actions to the ordering, programmers can verify that the changed program produces the right result.

However, the debugger cannot check if an ordering with added actions will deadlock.

A programmer must not be able to add actions to an ordering since a debugger cannot check that ordering. Without checking, the debugger cannot safeguard the programmer against orderings that deadlock. Even the canniest programmer needs this safeguard. The number of message paths to consider can be enormous. The programmer can easily miss one path among the multitude.

4.1.4 Promising to Always Ensure an Ordering

Always ensuring an ordering promises too much to programmers. The idea is simple: the programmer specifies a set of actions that he wants run in a certain order. The debugger is instructed to ensure that the actions in this set *always* run in a particular order.

Programmers are seduced by the up-side of this feature. By issuing a single command to such a debugger, the program behaves as if it has been fixed. That is, the programmer can see the outcome of a program change throughout an entire program run. The reordering technique allows a change during a small part of the program run; always ensuring an ordering would be much better. But the down-side of this promise makes this feature unattainable; unless all reordered actions always occur together, the debugger will deadlock.

The problem with this feature is deciding whether the set of actions has occurred. A simple example will show why. The programmer hypothesizes that actions **A** and **B** should always run in the order {**A**, **B**}. This control works well when actions occur soon after one another. But when **B** arrives and **A** does not, the debugger must wait indefinitely for **A**, or it will not ensure the order {**A**, **B**}. The debugger will deadlock if **A** does not arrive. To prevent deadlock, all elements of an ordering must always run together. Our intuition suggests that a set of actions will not always run together; examples in Chapter 5 provide evidence that confirms our intuition. Therefore, programmers should not have the power to instruct a debugger to always ensure an ordering.

In this pitfall, debugger developers promised too many features to the programmer. In the next section, they get into trouble by using a method that breaks one of their promises: not to

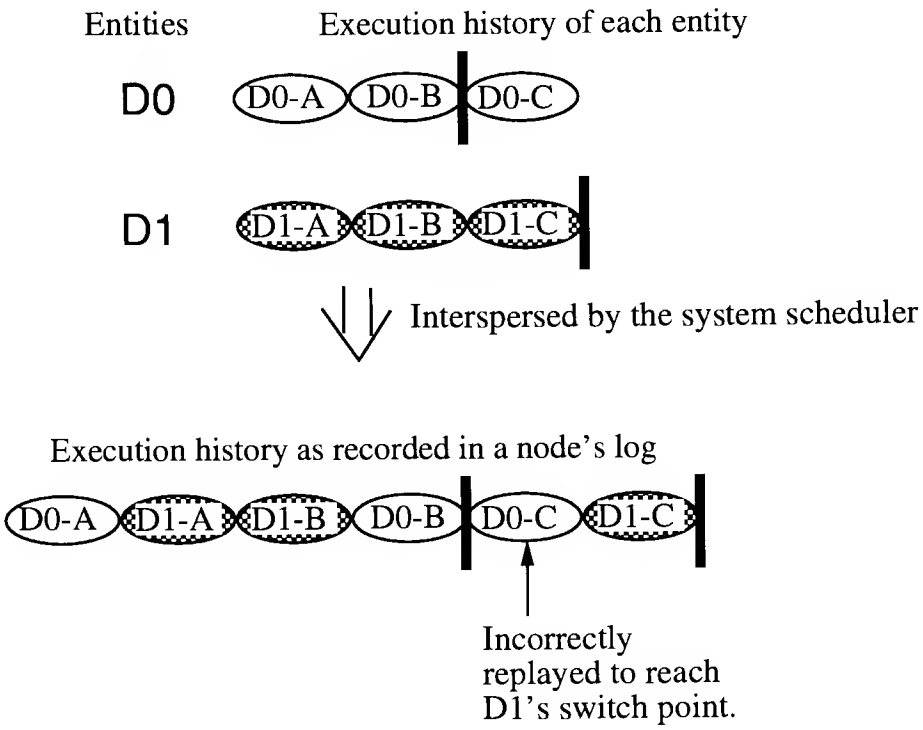


Figure 4-1: Different switch point locations

control actions after switch points.

4.1.5 Replay: Replaying Per Processor

Recall that Hindsight replays per entity. This method requires Hindsight to patch message and entity identifiers because Concurrent Scheme's runtime system interleaves actions from different entities. If Hindsight just replayed actions, it could replay per processor to ensure matching identifiers.

But this solution fails because entities switch to uncontrolled execution. The points at which entities switch can occur in different places in the log. This difference can result in replaying past one entity's switch point in order to reach another's. Figure 4-1 illustrates this possibility: entity D0 must replay past its own switch point in order to reach entity D1's switch point. This behavior causes trouble since the reordering debugger promises not to control actions after an entity's switch point.

4.1.6 Summary: Solutions to Avoid

Hindsight has avoided five pitfalls:

- switching all entities at once;
- reordering all concurrent actions;
- adding actions to an ordering;
- requiring a set of actions to always run in a certain order; and
- replaying per processor.

Still, Hindsight is not a perfect debugger. In the next section, we will learn how to refine Hindsight, to give it two new features: debugging longer-running programs, and ensuring that just the ordering is causing a different output.

4.2 Refining the Hindsight Design

This section will explain two ways in which Hindsight's design can be refined by future programmers. First, we will learn how Hindsight can be adapted to store smaller logs. Currently, Hindsight must store large logs since it records an event for every action. We will see how Hindsight can be adapted to record events selectively. Though this refinement restricts the orders we can specify, it allows the debugger to handle programs running for hours.

Second, we will learn how latest cuts prevent actions concurrent with the ordering from affecting program output. We will examine another alternative, *general cuts*, in which programmers specify a switch point for each entity. General cuts, though more powerful in theory than the latest cut, are impractical. An example will show us why reordering debuggers should provide the latest cut.

Before launching into detail, let us compare the relative merits of these two refinements. Selective logging is crucial since it allows Hindsight to handle much longer program runs. Latest cuts correct a deficiency in the earliest cut used by Hindsight. In sum, a reordering debugger

with these features is a powerful tool, capable of handling long-running programs while ensuring that only the specified ordering affects program output.

4.2.1 Logs: Netzer's Breakthrough

Large logs generally hold useless information. Their size makes it nearly impossible to replay or reorder some programs. Hindsight records large logs because it records an event for every action.

In 1992, Professor Robert Netzer set out to reduce log sizes. He built a replay debugger that records events for certain actions. His results have been superb; his debugger reduces log size by two to three orders of magnitude.[1] Hindsight can use this same technique to record smaller logs.

Let us examine how Professor Netzer saves space. His key insight is that many consecutive actions can occur in only one order. Like railroad cars, one action follows the next, from engine to caboose. When the engine hooks onto the first passenger car, all passenger cars follow in order. When the last passenger car hooks onto the first freight car, all freight cars follow in order. The order in which the first and last cars from a group hook determines the order of all train cars. As with train cars, so with actions. The debugger must ensure two things: that it hooks onto the right chain of actions; and that it starts no other chain until it finishes the current one.

Netzer's logging method is illuminated by the following example (Figure 4-2). We focus on the actions from entity D1: **A**, **C**, **E**, and **F**. Netzer's logging method records events for **E** and **F** only. Here's why: since **A** is the first action, it must run in that order. The debugger records nothing. When **C** arrives, the debugger determines that **C** happens after (in Lamport's sense [Lam78]) the previous action, **A**, so again it records nothing. **E** follows, but after **C**, so once more the debugger records nothing.

But when **F** runs, the debugger determines that **F** does not happen after **E**. Therefore, it

[1] Dr. Netzer's technique works well on a small number of processors. It does not scale to many processors, but this limitation scarcely matters since programmers debug almost all programs on only a few processors.

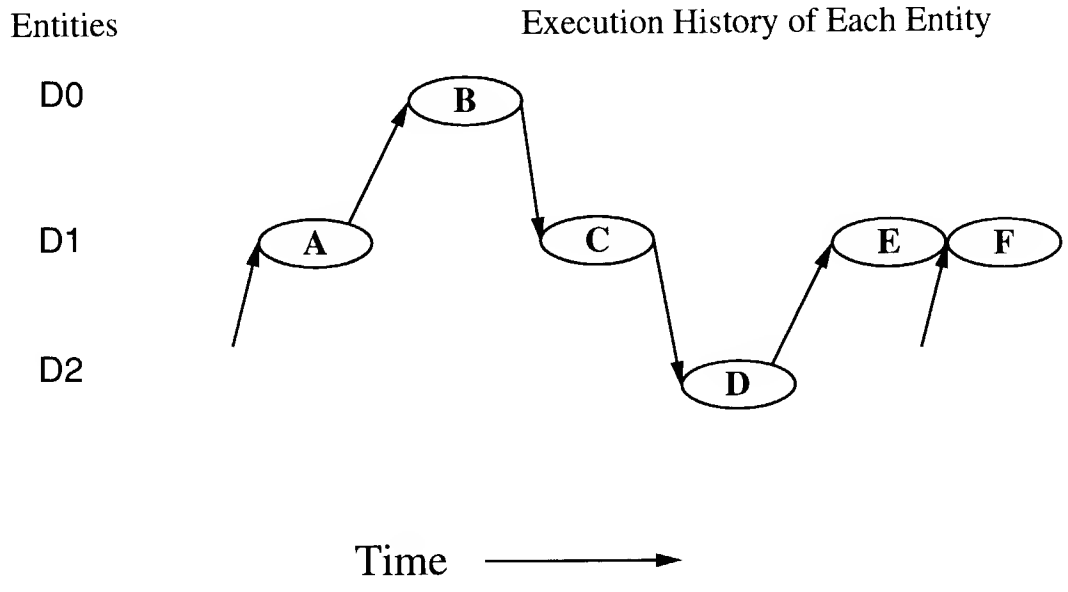


Figure 4-2: Example showing the advantage of Netzer's approach

records the order {**E**, **F**}.[2] Netzer's logging method has reduced the number of actions recorded in this example by a factor of two. Real examples use this same technique to compact chains of hundreds or thousands of actions into one log event.

Though smaller logs are a strength, they also restrict programmers to reorder just the recorded actions. Let us find actions in the last example that the debugger might want to reorder, but cannot. Action **F** can run before any of three actions: **A**, **C**, or **E**. But Hindsight records just **E** and **F**, so the programmer can reorder just **E** and **F** at first. To run **F** before **A** or **C**, the programmer must reorder multiple times. For instance, running **F** before **C** is a two step process. The first step is to run **F** before **E**, then **F** before **C**.

Clearly this logging technique has both strengths and weaknesses. The strengths far outweigh the weaknesses. The next logical question is: how complex is it to build a reordering debugger with Netzer's logging technique?

[2]Netzer's published algorithm records just the second action, **F**, but we do not discuss this efficiency hack.

Building a Reordering Debugger with Netzer's Logging Technique

Building a reordering debugger with Netzer's logging technique is simple. *Dependence vectors* are the crucial ingredient. A dependence vector is one entity's view of what has already occurred in the system at a certain instant. Its view, its dependence vector, changes each time it receives a message. Element i of a dependence vector holds the last known action by entity i. For better or for worse, these elements of dependence vectors become the lifeblood of the reordering debugger.

Debugger components use the dependence vectors to check action orders and to find the latest switch cuts. The replay component now controls the order of only certain actions on controlled entities. The key to this different type of controlled execution is to order the beginning and end of chains of actions. That way, all chains before an ordering occur in the same order. Using Netzer's technique does not complicate debugger components aside from logging; they are just different.

The only drawback is the size of dependence vectors. For parallel machines with many processors, they are quite large. Large dependence vectors cause trouble since they consume vast amounts of network bandwidth. The debugger must append a dependence vector to every message. When machines are large, the size of dependence vectors can dwarf the size of messages. This results in vastly slowing down a running program since the network requires a lot of time to transmit messages with dependence vectors.

Despite all that, the future of Netzer's technique is bright. Programmers rarely develop a program on large machines; they almost always develop it on small machines. As mentioned, that is where Netzer's technique works well.

4.2.2 Switch Cut: Use the Latest One

Programmers must use the latest cut so that concurrent actions do not affect program output. If a programmer uses a cut other than the latest one—that is, one that does not control all concurrent actions—he cannot be sure whether the ordering or a different order of uncontrolled, concurrent actions produces an incorrect result. By contrast, the latest cut replays all actions

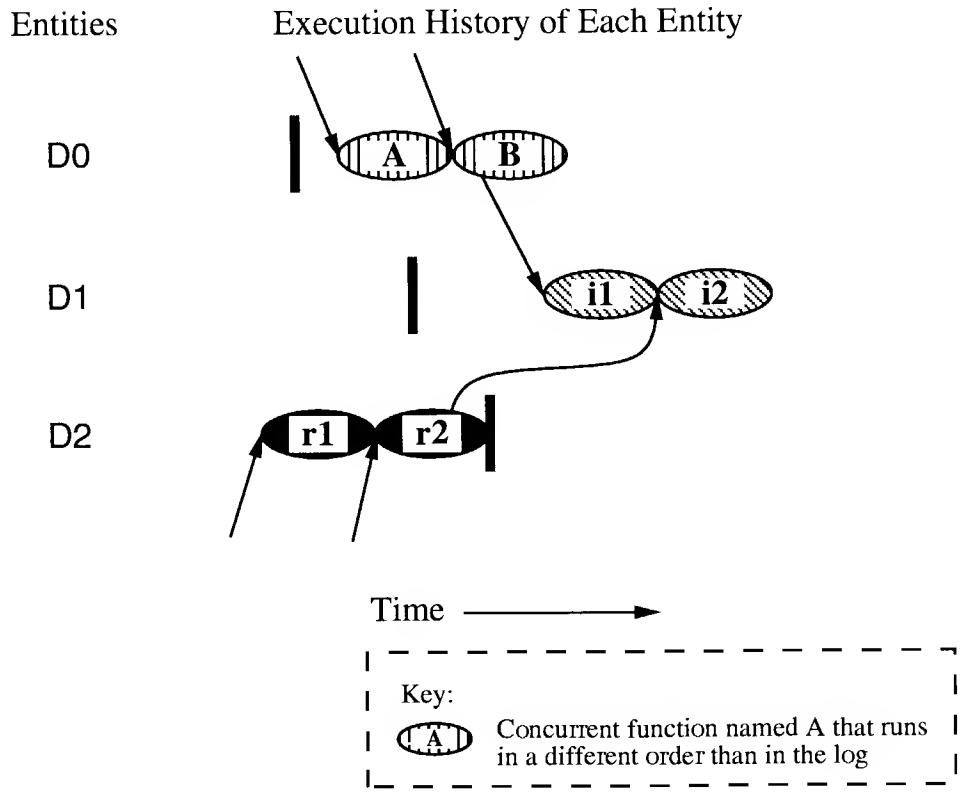


Figure 4-3: Concurrent function execution order matters

concurrent with the ordering. This ensures that only the ordering will modify variable values differently.

Consider an example where concurrent actions can change program output. Figure 4-3 shows how the order of concurrent actions **A** and **B** indirectly affects the variable values in entity D1. The indirection is action **B** starting action **i1** on entity D1. Action **i1**, in turn, can change entity D1's variable values, used by **i2**. Trouble arises if **A** and **B** run in the reverse order. **B** might not start **i1**, so **i2** uses different variable values, resulting in a different program output. Thus, the order of concurrent actions **A** and **B** can affect program output.

As mentioned, a general cut is useful in theory. It provides control over switch points since the programmer specifies every one. But specifying each switch point can be cumbersome. Indeed, Goldberg et al. [GGLS91] have suggested that the programmer specify just some switch points and then instruct the debugger to find the rest. This change makes the general

cut easy to specify.

Ease of use aside, the general cut squanders the reordering technique's most important benefit: control of action order. To preserve benefits of the reordering technique, the debugger must use the latest cut. But what about stopping in places besides the latest cut? This question raises an important issue. Where to stop a program execution and how long to control it do not affect one another. If the programmer wants to examine the program before the latest cut, he can. But that has no bearing on what part of a program execution the debugger should control.

While using the latest cut eliminates different ordering of concurrent actions, there remains an unsolved problem related to uncontrolled orderings. The problem is that actions after the ordering can still run in a different order. Indeed, if the programmer thinks an error occurs too early, he can hide the error using the reordering technique. Let us see why hiding an error is most likely a rare occurrence. Programmers do not think errors occur earlier than they really do since no symptoms can occur then. Instead, programmers find symptoms long after errors really occurred. Thus, programmers are far more likely to think errors occurred later than they actually did. A reordering debugger will replay errors before the ordering, so the error is unaffected by action order after the ordering.

Providing the Latest Cut

We can now see that the latest cut adds another chore to the checking component. Hindsight still needs the earliest cut to check the ordering. Now it must also find the latest cut. The implementations are quite similar, so providing it is trivial; instead of moving backwards, the latest cut calculation moves forward in the log.

4.2.3 Improving the Reordering Technique: A Summary

The reordering technique can be improved in two ways:

- by reducing the log size, and
- by providing the latest cut.

Professor Netzer's logging technique is the more important refinement. Though it restricts the possible orderings, programmers should implement it since a reordering debugger that provides it can handle very long running programs. With the latest cut refinement, a reordering debugger provides program outputs unaffected by concurrent actions.

4.3 Summary

This chapter has sorted out the desirable from the undesirable alternatives to the Hindsight design. The five undesirable alternatives are presented here to warn developers. Two of the five cause inefficiency—switching all entities to uncontrolled mode together and reordering all concurrent actions. The first slows the computer; the second slows the programmer. The other three undesirable alternatives lead to deadlock—replaying per processor, adding actions to orderings, and always ensuring an ordering.

The two desirable alternatives serve as models for developers. Of the two, Professor Netzer's logging method is the more important since it greatly reduces log size. The latest switch cut helps by keeping concurrent actions from changing the outcome of an ordering. The experience with Hindsight has not shown this to be a problem; still, developers should provide the latest cut to assure that programmers have untampered orderings.

Now that we have seen the Hindsight design and its alternatives, we ask another question: how well does the reordering technique perform? The next chapter will answer this question and describe lessons learned from the reordering of three real programs.

Chapter 5

Experience

In Chapter 2, we learned how to use the reordering technique. This technique allows a programmer to arrange actions to occur in a certain order by changing the log of a program execution. In Chapter 3, we learned how to implement the reordering debugger, Hindsight.

In this chapter, we will use Hindsight to evaluate the reordering technique. We will arrange a new action order in three programs: the Proteus simulator's runtime system [BDCW92]; a non-replicated, concurrent *Balanced Tree* (B-tree) algorithm [LS86]; and a replicated, concurrent B-tree [JC92]. We use Hindsight to test ideas about these errors.[1] These three programs were chosen to answer four questions:

- How many actions must a programmer reorder?
- Does the reordering technique identify real bugs?
- How could the reordering technique be improved?
- How useful is it to focus on one entity alone?

The runtime system example [BDCW92] answers all four questions. In this example, an error occurs when an interrupt action occurs between two other actions. The programmer need

[1] Hindsight was not used to find ordering errors since that is not its purpose. Many tools exist that find possible ordering errors [Tay83, EP89, CKS90, EP89, Sch89, DS90, NM92].

modify the order of just three actions: the two other actions and the interrupt action. This example shows that programmers can test hypotheses about the cause of real bugs by changing the order of actions on just one entity. It also reveals two ways in which the reordering technique can be improved: by running a modified program with the original log; and by reordering non-contiguous sets of actions. Before adding these refinements, the reordering technique is useful; after adding them, it will be even more so.

Together, Lanin and Shasha [LS86] have developed a non-replicated, concurrent B-tree algorithm. In Lanin and Shasha's algorithm, an error occurs when split and merge operations occur in one order on a B-tree node, but in the opposite order on the node's parent. As above, the error is restricted to a few actions on one entity. In this example, programmers benefit from focusing on one entity, rather than all concurrent actions. This example shows not shortcomings but more potential uses of the reordering technique. While debugging, the program was reordered to cause an error. Thus, the reordering technique also promises to be a useful part of a testing tool.

Both the Lanin and Shasha error and the runtime system error happen on unreplicated data. In these examples, a programmer should focus on one entity when forcing an action order. But our third example is different. Johnson and Colbrook's B-tree algorithm [JC92] uses multiple copies of B-tree nodes. Because of an ordering error, the copies do not have the same information. Again, programmers benefit from focusing on one entity. As before, programmers can identify real bugs by changing the order of only a few actions. Unlike before, programmers see a need for a tool to display logs. However, this shortcoming stems from our implementation, not the reordering technique. Programmers can test hypotheses about errors, even on data that resides on multiple nodes.

These examples show that the reordering technique helps programmers debug programs. In fact, we have also learned that the reordering technique could serve as the basis for a testing tool, an added benefit. Along the way, we have learned two other important lessons. First, programmers need to specify more orderings than Hindsight allows. Second, a crucial debugger feature is a good user interface for displaying logs. The ordering debuggers that incorporate

these lessons will prove invaluable to parallel programmers.

5.1 A Runtime System Error

For the first experiment, we have implemented an erroneous part of the Proteus runtime system. Proteus [BDCW92] simulates a parallel computer by running many virtual processors on one physical processor. In order to spend no time simulating idle processors, Proteus assigns a mode to each processor. Processors at work are in the **active** mode; those at rest are in the **idle** mode.

An ordering error occurs when an interrupt coincides with a processor switching from active to idle mode. More interrupts must still be run, but the processor does not detect them. This error develops when an interrupt occurs after the processor checks for waiting interrupts, but before it switches to idle mode. Since processors do not run interrupts in idle mode, the waiting interrupts are delayed until the processor becomes active again.

In Proteus, the interrupt handler routine is a loop. The body of the loop runs a waiting interrupt; if another waiting interrupt exists, the loop repeats. While Proteus runs an interrupt in the loop body, it queues other interrupts. If the processor detects no more waiting interrupts, it calls `make-idle`, changing the processor mode to idle. Figure 5-1 shows the pseudocode for this loop.

An ordering error occurs when an interrupt happens between the end of the loop and the call to `make-idle`. Proteus will enqueue the interrupt, but the processor switches to idle mode straightaway. This switch greatly delays when the interrupt will run.

Figure 5-2 shows the Concurrent Scheme code for the erroneous part of the Proteus runtime system. We implement the interrupt handler with the functions `check-interrupt` and `handle-interrupt`. To create a loop, these functions call one another. If there are no waiting interrupts, `check-interrupt` exits the loop by calling `make-idle`. The functions call one another with **make-thread**, a construct that starts a function using an asynchronous message send. An unwanted interrupt can occur before `make-idle` because `check-interrupt` calls `make-idle` asynchronously. Like the original code, `make-idle` changes the processor state to **idle**. However,

```
while (waiting interrupts exist)
      handle_interrupts();

/* INTERRUPT CAN OCCUR */

make-idle();
```

Figure 5-1: Pseudocode of the Proteus interrupt handler

it also identifies errors with printed error statements.

The runtime system error is created by running four actions in one domain: `handle-interrupt`, `check-interrupt`, `enqueue-interrupt`, and `make-idle`. Many actions run in this domain, but let us concentrate on the last three: `check-interrupt`, `enqueue-interrupt`, and `make-idle`. `Make-idle` prints an error message since interrupts are waiting when the processor becomes idle.

A potential error seems to occur when `enqueue-interrupt` runs between `check-interrupt` and `make-idle`. But it is unclear whether this ordering causes the error. There are three other likely culprits: a memory-overwrite bug; a faulty conditional in `check-interrupt` that calls `make-idle` under the wrong conditions; or a faulty conditional in `make-idle` that prints an error message when no error exists.

To gauge whether this ordering error is the real culprit, we run `enqueue-interrupt` after `make-idle`. This reordering obeys the rules. Because it does, Hindsight runs the program with debugger control. This ensures that the last two functions run in the order {`make-idle`, `check-interrupt`}. During the reordered execution, `make-idle` signals no error. This result is good. We have found the culprit.

5.1.1 Lessons from the Runtime System Error

Changing this ordering error in Proteus's runtime system clearly shows the value of the reordering technique. Although this example may appear overly simple, almost contrived, it is typical of real ordering errors. The reason to provide the reordering technique is the difficulty

```
(check-interrupt
 (lambda (dom)
   (if (equal? (length queue 0) 0)
       ;; IF NO MORE INTERRUPTS, ASYNCHRONOUSLY CALL MAKE-IDLE
       (make-thread make-idle '() :domain dom)
       ;; IF MORE INTERRUPTS, HANDLE ONE
       (make-thread handle-interrupt (list dom) :domain dom)
       )))

(handle-interrupt
 (lambda (dom)
   ;; SYNCHRONOUSLY RUN THE FIRST JOB IN THE INTERRUPT QUEUE
   (run-job (car queue))
   ;; DELETE THE JOB THAT WAS JUST RUN
   (set! queue (cdr queue))
   ;; CHECK FOR MORE INTERRUPTS
   (make-thread check-interrupt (list dom) :domain dom)
   ))

(make-idle
 (lambda ()
   ;; CHECK IF AN ERROR OCCURRED
   (if queue
       (print "****error: non-empty queue when going idle")
       (print "going idle with an empty queue"))
   ;; SET THE PROCESSOR'S STATE TO IDLE
   (set! processor-state idle)
   ))

(enqueue-interrupt
 (lambda (job)
   ;; ADD ANOTHER JOB TO THE INTERRUPT QUEUE
   (set! queue (cons job queue))
   ))
```

Figure 5-2: Interrupt handler in Concurrent Scheme

of repeating such errors.

In this experiment, we had to reorder just two actions to fix this error. This experiment suggests that real ordering errors consist of just a few actions. Programmers can easily manage the two possible orderings. This example also shows two ways in which it appears we can refine the reordering technique. We could run a modified program with the original log. Or we could allow the programmer to reorder a non-contiguous set of actions.

Running a Modified Program with the Original Log

Ideally, programmers would like to force an order on changed programs. That way, they can test whether a code change fixes a bug. But the debugger is only an agent. It blindly controls programs according to a log. If the changed program sends different messages, the debugger will deadlock while trying to replay it. We believe that changed programs often send different messages. If a program lacks synchronization, the programmer adds messages to provide it.

The runtime system, for instance, sends different messages when the programmer fixes it. To do so, he changes **check-interrupt** so that it calls **make-idle** with a remote procedure call instead of an asynchronous message send. As described earlier (Section 3.1), a reply from a remote procedure call always runs an action; an asynchronous call lacks the reply message needed to run such an action. Thus, **make-idle** sends a message in the changed program that it did not send before. Figure 5-3 illustrates this change. Although we have presented this example in terms of Concurrent Scheme, it is true for any language.

Specifying a Non-Contiguous Set

Programmers want to be able to impose an ordering on a non-consecutive set of actions. In Figure 5-4, a programmer, wanting to reorder actions **A** and **C**, chooses the contiguous set {**A**, **B**, **C**}. But when he does, the debugger signals an error since the set contains both **A** and **B**. **A** sends the message that starts **B**, so **B** depends on **A**.

The programmer cannot reorder actions that other reordered actions start. This happens frequently. It occurs when reordered actions call an action with either the apply-within-domain

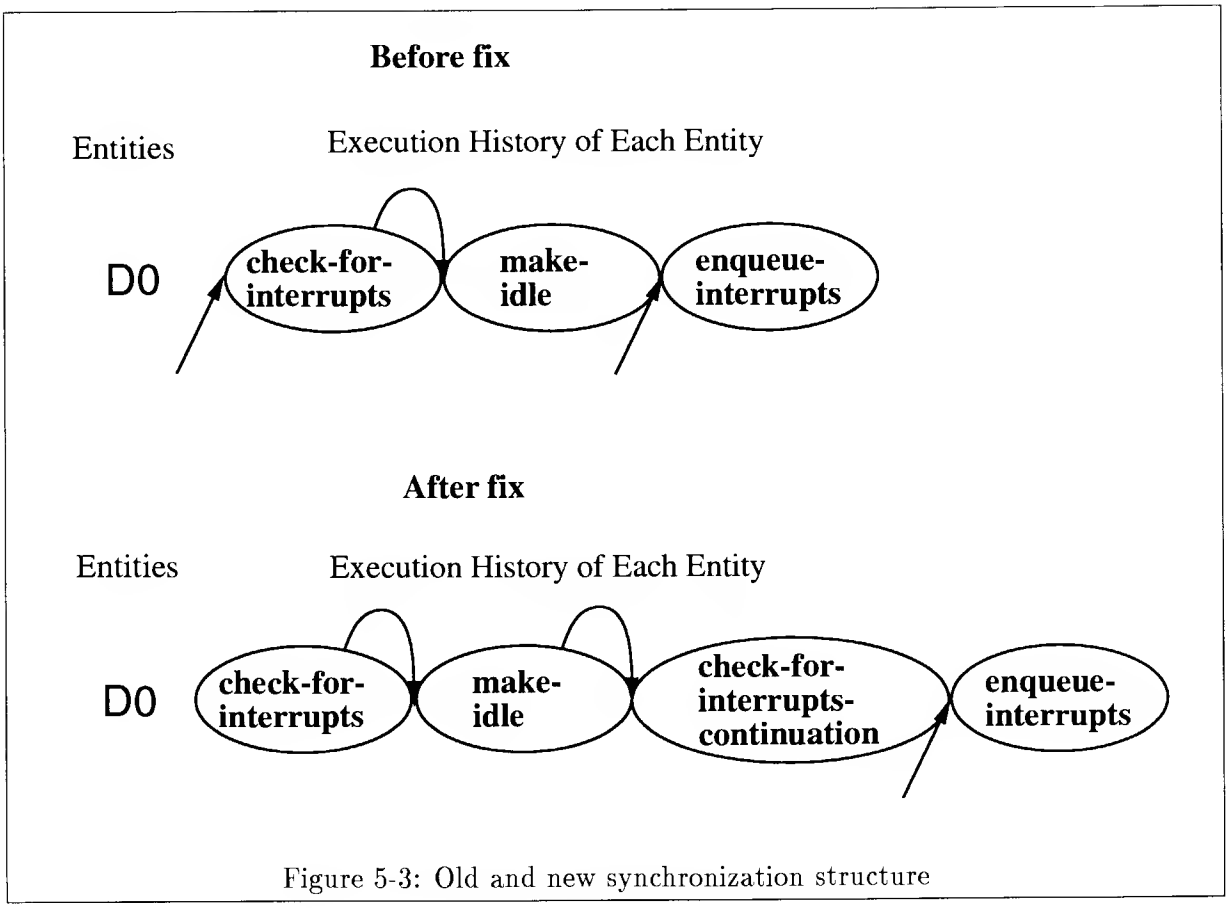


Figure 5-3: Old and new synchronization structure

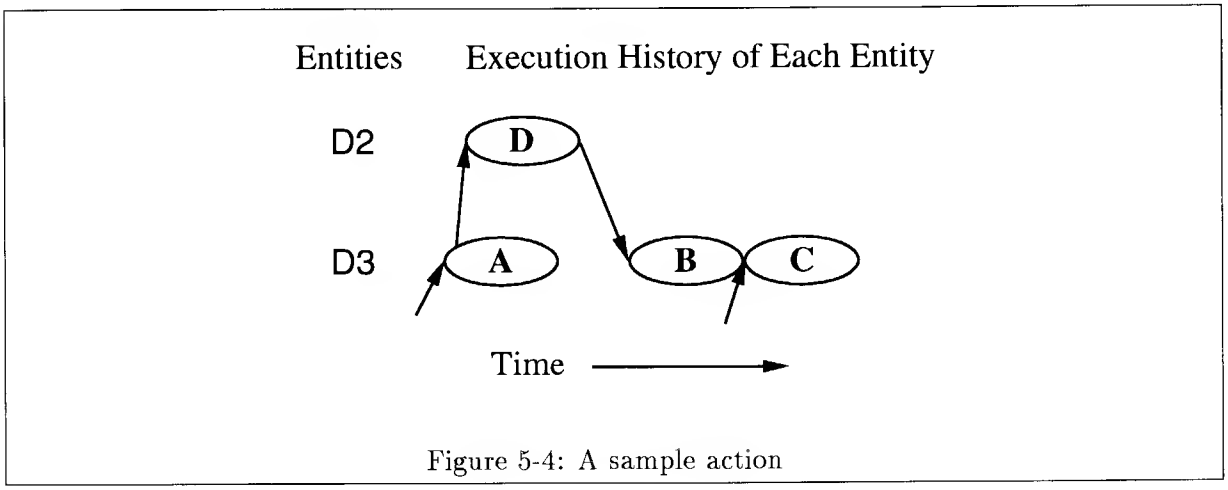


Figure 5-4: A sample action

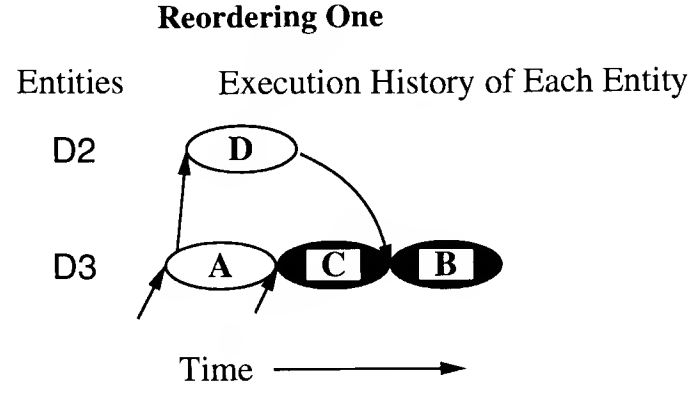


Figure 5-5: Step one to reordering **A** and **C**

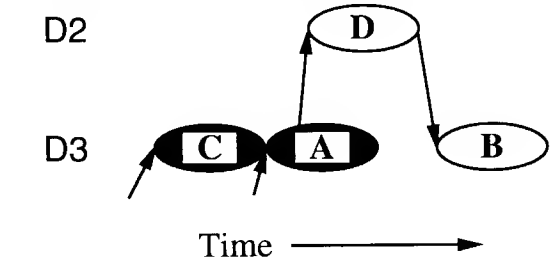


Figure 5-6: Step two to reordering **A** and **C**

or delegate constructs. These constructs provide blocking and non-blocking remote procedure calls, respectively. The reply of the remote procedure call runs an action in the calling entity.

In the prior example, running action **A** after **C** requires two steps. Figures 5-5 and 5-6 show these steps. In step one, the programmer changes the order of just **B** and **C**. In step two, the programmer uses the log from the reordered execution to change the ordering of just **A** and **B**.

The first ordering, running **B** before **C**, changes the meaning of Concurrent Scheme constructs. No action should run between the call and reply of a blocking remote procedure call. But the log does not include this information. The debugger allows the programmer to specify this ordering.

This problem applies to any language. Logs will generally obscure the meaning of some

construct. Without proper data, the debugger will allow action orders that language constructs prevent. A reordering debugger should disallow such orderings, but how? A debugger can use one of two means: recording data about constructs; or including filter programs that deduce constructs from the log.

5.2 B-Tree Examples

B-trees maintain data in a tree format. The main advantage of this format is that it allows programs to access data quickly (in O(log N) time). In 1972, Bayer and McCreight created this data structure for a specific task: tracking the sectors in which a file resides [BM72]. Since then, B-tree's have been used far more broadly.

A B-tree is a tree of nodes. Each node has one parent and some children. In the entity model, each node is an entity. The parent node is one step closer to the root of the tree; each child is one step further away. Such trees are called balanced because every leaf is the same distance from the root. In these examples, we consider only *B+-trees* [Knu73], which store all data in their leaves. A B-tree stores data in pairs: each pair consists of a key and a datum. The leaves contain these pairs. In the file system application, the keys are file names, and the data are the sectors in which the file resides. The internal nodes contain a different kind of pair: a pointer to a child node and a key. Internal node keys direct all B-tree operations to the correct child.

A B-tree provides the programmer with the operations **lookup**, **insert**, and **delete**. All operations start at the root of the tree and move toward its leaves. An entity-model action in these algorithms is an operation's uninterrupted computation on one node. Operations move from root to leaf by key values in the internal nodes, as we cross a stream by hopping from one stone to the next. To keep the B-tree balanced, the **insert** and **delete** operations call **merge** and **split** operations. These operations move from a leaf toward the root, changing node contents to maintain a constant distance from root to leaf. To each node that it visits, a **merge** operation removes a pointer and a **split** operation adds a pointer. Both operations stop moving up when the tree is balanced. They can stop before reaching the root.

B-trees are now used in many other applications, including parallel ones. Researchers have developed parallel B-tree algorithms to process many operations at once. To ensure that operations access a node without interruption, parallel B-tree algorithms must include synchronization. However, algorithm designers sometimes omit necessary synchronization. Errors occur. To fix them, programmers must change the order in which actions run. Programmers must use the reordering technique to try different orderings without fear of hiding an error.

5.2.1 An Unreplicated B-tree Error

Lanin and Shasha's algorithm locks only a parent node as operations pass up the tree. Lanin and Shasha had forgotten to synchronize consecutive **merge** and **split** operations. If the number of entries in a node grows and shrinks quickly, a **merge** can closely follow a **split**. Lanin and Shasha's algorithm allows a later **merge** to run on a parent node before the previous **split**.

An error occurs when a **merge** precedes the previous **split**. The **merge** operation deletes the pointer that the **split** operation tries to split. Thus, a node exists in the B-tree without a pointer from its parent. The parent routes an operation to the wrong node, and performance suffers. The operation then moves to the missing node, but not before the parent misroutes it.

Let us assume that a **split** and **merge** operation both run on the same leaf, **split** followed by **merge**. The two operations start **complete-split** and **complete-merge** operations on the parent. An error occurs if **complete-merge** runs before **complete-split** on the parent. If so, **complete-merge** has already deleted pointer, P, when **complete-split** tries to split P into two pointers.

We have implemented this ordering error in Concurrent Scheme using six functions: **insert**, **delete**, **split**, **merge**, **complete-split**, and **complete-merge**. The implementation consists of one parent node and its three child nodes. Each node runs in its own domain.

Though an error exists, the algorithm runs correctly: **complete-merge** runs after **complete-split**. We first impose an incorrect order, then test hypotheses about it. Using Hindsight to impose an incorrect order shows that it can serve as a component of a testing tool. Hindsight's ability to run one ordering at a time, however, is too primitive to be a testing tool. It is like

```
3. complete-merge
1. complete-split
2. print-state
```

Figure 5-7: Error producing ordering

node name	elements
Node 1	24 56
Node 3	128 144

Table 5.1: Node 2 is incorrectly missing.

an assembly language instruction: a building block, not a panacea.

To produce the ordering bug, we run **complete-split** after **complete-merge**. To see the result, we move the **print-state** after both **complete-split** and **complete-merge**. Figure 5-7 illustrates this ordering. When the debugger runs this order, the parent node lacks a pointer to node 2, as shown in Table 5.1.

Now we pretend that the erroneous order had occurred initially. We start with the log of the modified execution and reorder it. We hypothesize that the problem lies in the order of the last three actions running on the parent node. We then tell the debugger to run **complete-merge** after **complete-split**. This order runs correctly, as shown in Table 5.2.

Lessons from Lanin and Shasha's Error

Like the previous example, this one also suggests that ordering errors contain a manageable number of actions. We had to rearrange just three actions in this example. This example also

node name	elements
Node 1	24 56
Node 2	96 120
Node 3	128 144

Table 5.2: Correct ordering includes node 2.

shows Hindsight's ability to reorder repeatedly a program execution. This ability helps with two tasks: debugging multiple errors and producing an error.

We did not expect to produce errors with the reordering technique. However, this technique seems a valuable component of an adequate testing tool. But as mentioned, our technique by itself is not a testing tool, since a programmer imposes just one ordering at a time. To test real parallel programs, programmers must specify thousands of orderings, or more. Thus, a testing tool must also include a component to specify sets of orderings.

As for refinements, the function names **complete-split** and **complete-merge** in this example reveal one way to improve Hindsight. To reorder, programmers must know what pointers these operations split or merge; Hindsight shows only their names. We have constructed this example so that only one child splits and merges. If others also split or merge, reordering would be very difficult. Fixing this problem, however, requires recording more data. But since we want to minimize log size, displaying more data is possible only if Hindsight records fewer events.[1]

These lessons reinforce those from the first example. The third example will provide more evidence, but highlight different features of reordering debuggers.

5.2.2 A Replicated B-tree Error

In 1992, Johnson and Colbrook jointly developed a B-tree algorithm. Their algorithm tries to process many operations at once by using many copies of each B-tree node. All copies of a node have the same data. We call these copies *replicas*, and the whole tree a *replicated B-tree*.

Johnson and Colbrook had to balance their desire for speed and for consistency. They chose speed; as a result, their algorithm allows copies of the same node to have different data. Paul Cosway found this error in their algorithm while researching B-tree algorithms.

Johnson and Colbrook's algorithm keeps nodes consistent asynchronously. Let us consider an **insert** operation. It starts at the root node and moves down the tree following pointers in the intermediate nodes. Eventually, it finds a copy of the node where the new key should

[1] See Professor Netzer's logging method, Section 4.2.1

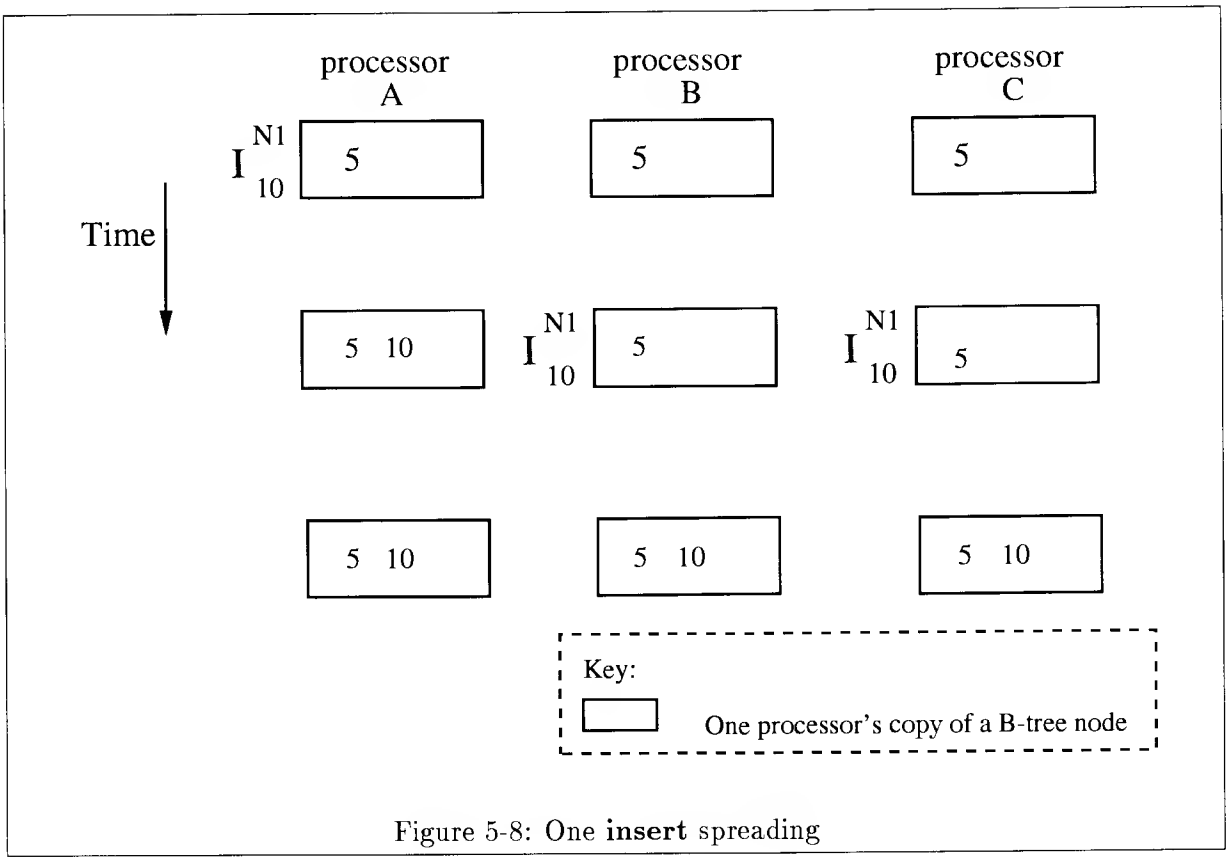


Figure 5-8: One **insert** spreading

reside. The **insert** occurs on that copy. This copy then sends the new data to the other copies. Figure 5-8 shows what happens once the **insert** reaches the copy of N1, where key 10 should reside, on processor A. The operation first updates processor A's copy of N1, and then asynchronously inserts the same element in the copies of N1 on other processors.

Two **insert** operations can occur in different orders if they start on different processors. Figure 5-9 shows the result of inserting elements 10 and 15 simultaneously, on different nodes.

Insert operations can also overlap with **split** operations. **Split** operations spread just like **insert** operations: the **split** first occurs on one copy of the node and the copy sends the **split** command to other copies asynchronously.

Unfortunately, overlapping **split** and **insert** operations cause ordering errors. Consider the example in Figure 5-10, focusing on processor A. In this example, the propagating **split** and **insert** operations occur in the order {**split**, **insert**} on processor A. On other nodes, they are reversed: {**insert**, **split**}. The result, shown at bottom left in this figure, is an invalid state:

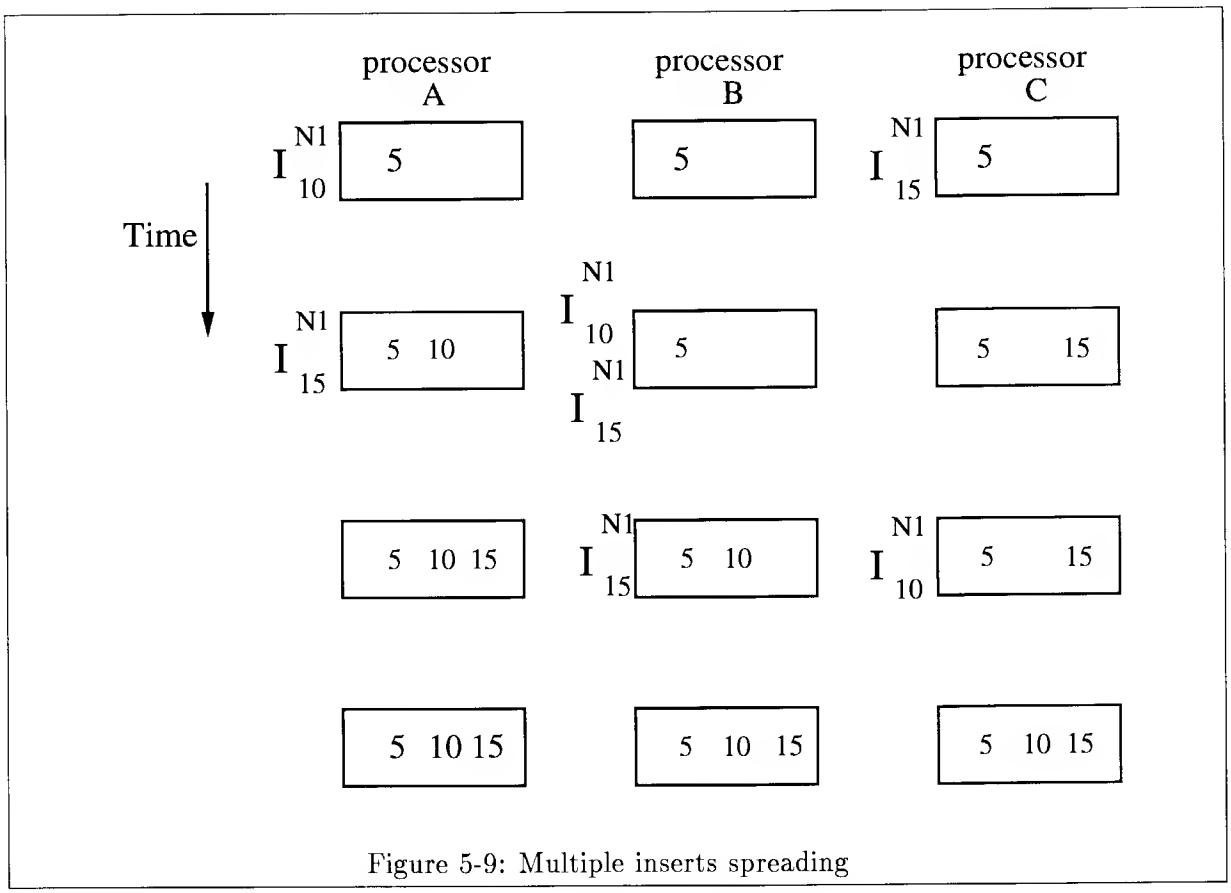


Figure 5-9: Multiple inserts spreading

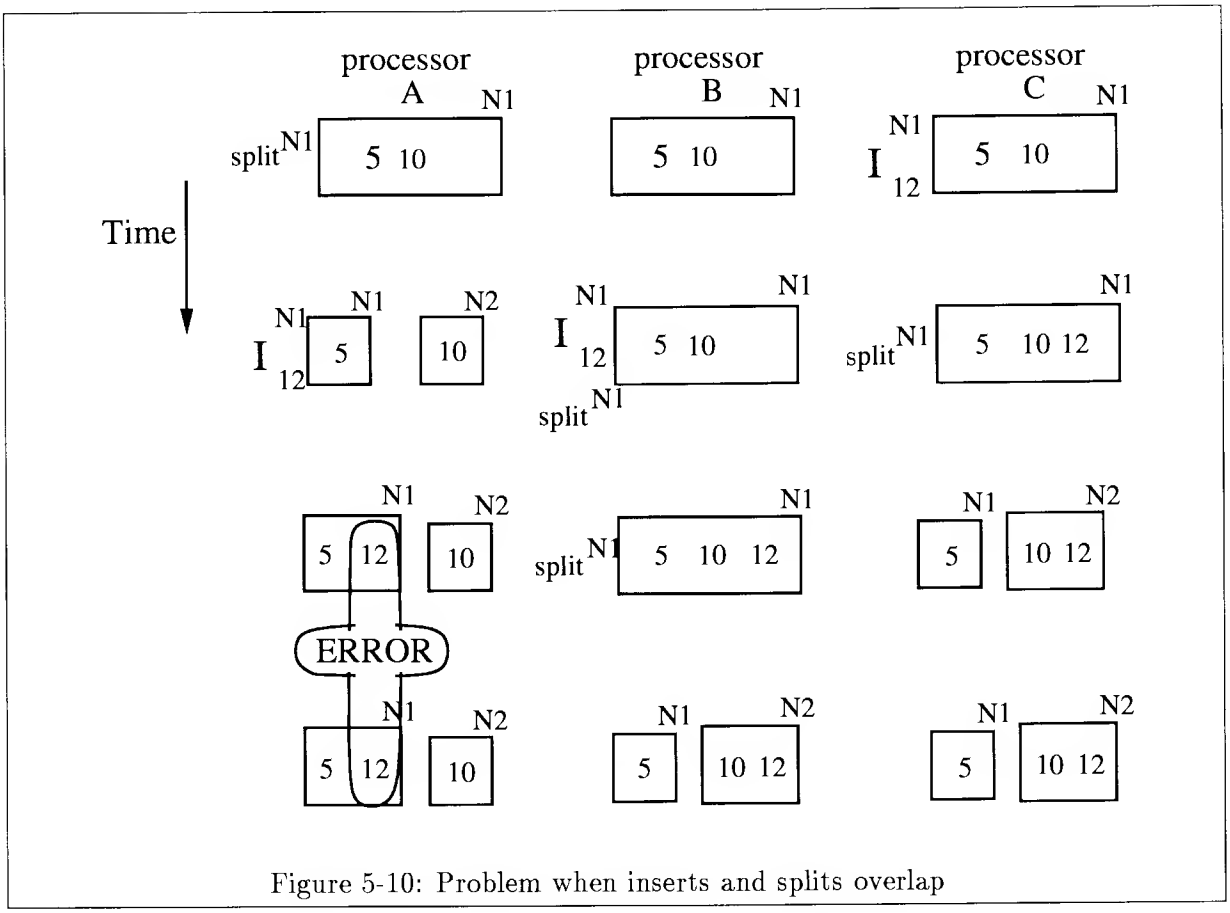


Figure 5-10: Problem when inserts and splits overlap

processor A's elements have a larger key, 12, inserted before a smaller key, 10.

Examining this execution, we hypothesize that the program may be caused by the order of **split** and **insert**. To test this hypothesis, we reorder the log. We run the **split** and **insert** operations on processor A in reverse order: {**insert**, **split**}. Run in this order, the program produces the expected result—element 10 resides in N1 and 12 resides in N2. We have found an ordering that works by reordering the program. Without the reordering technique, the error could have disappeared.

Lessons from Johnson and Colbrook's Algorithm

Like the other experiments, this one has shown that the reordering technique is useful for testing hypotheses about ordering bugs. It indicates that real ordering errors consist of just a few actions—in this case just three. Reordering this error also demonstrates that the programmer

needs to focus on a small part of this execution to avoid being swamped with extraneous log data. A B-tree with thousands of nodes could drown programmers in excess log data.

5.3 Summary

This chapter showed programmers using Hindsight to test hypotheses about ordering errors. We have also seen three ways to improve the reordering technique. First, a programmer would like to impose orderings on non-contiguous sets of actions. Second, a debugging system should combine a mechanism like Hindsight with tools that can display logs graphically. Third, a filter tool would be useful to identify potential errors.

It is clear from this experience that the reordering technique is useful. To affect a change, programmers must change the order of just a few actions. They are best off changing one entity alone; they can then ignore extraneous log data. Together, these benefits show how programmers would benefit and thus why developers should include it in future parallel debuggers.

Chapter 6

Conclusions

In this thesis, we have learned about three things. First, we learned of the *reordering technique.* Second, we learned of a reordering debugger, Hindsight. Third, we learned of experience using Hindsight to debug real programs. This experience has shown us that the reordering technique is valuable for debugging ordering errors.

We first learned of the need for the reordering technique in Chapter 1. Using current debuggers, programmers must change the program itself to fix an error. But when a programmer runs a changed program, trouble occurs; the debugger cannot replay changed programs, so the program might not reach the same environment that first caused the error. As a result, the program might not run the error-causing actions. When it does run these actions in a correct order, the programmer cannot even tell if his change deserves the credit. Actions can run in correct order by chance, due to fragile system timings. Such timings can easily change, bringing back the error that the programmer had thought was fixed.

Second, Chapter 2 showed us how to use the reordering technique. A programmer uses this technique to help fix errors caused by lacking synchronization. The programmer specifies a new order of actions in a log of a particular program run. Using this modified log, a reordering debugger ensures that the program's actions run in the specified order. In this manner, the programmer can specify different orders until he finds one that works. The programmer can then return to the original program and add the missing synchronization. The reordering

technique therefore enables a programmer to fix ordering errors.

Third, we have learned how to implement a reordering debugger. Chapter 3 describes Hindsight, the debugger implemented for this thesis. This design shows the key aspect of reordering debuggers: switching from controlling actions to not controlling actions. We have learned that debuggers must switch independently at each entity, without waiting for other entities. We also know from Chapter 4 that reordering debuggers must switch entities to uncontrolled mode at the latest switch cut. This cut prevents different concurrent orderings from affecting program output, but does not restrict where programmers can stop and probe the program.

We met five main pitfalls that debugger developers should avoid in Chapter 4. For example, programmers must not add actions to an ordering. Extra actions prevent the debugger from checking whether it can run the ordering, resulting in possible deadlock. In sum, the pitfalls described in Chapter 4 steer programmers clear of seductive solutions whose side effects range from inefficiency to deadlock.

Chapter 5 describes our most important lessons, learned from the reordering of three real programs. Recall that Hindsight is the first parallel debugger to provide the reordering technique. Likewise, this thesis is the first evaluation of how this technique performs. From our experience, we have learned that programmers can gauge whether certain orderings do indeed fix errors. They can do so by focusing on one entity alone. Programmers need change the order of just a few actions on this entity. There are a manageable number of orderings for programmers to consider, making the reordering technique easy to use. These lessons imply that the reordering technique is a good tool for fixing the notoriously difficult ordering errors.

The reordering technique is good, but we have learned three ways to refine it further: first, using Netzer's method to record much smaller logs; second, reordering non-contiguous sets to allow more orderings; and third, using the latest cut to gauge the effect of the reordered actions alone. While all three refinements are important, Netzer's method for recording smaller logs is the crucial one.

In sum, we have learned many lessons. We have learned that current debugging techniques

cannot handle errors caused by a program lacking synchronization. We have learned about the reordering technique, which can handle these errors. We have also seen how to build the reordering debugger, Hindsight. We have learned that the design of Hindsight includes good decisions that other developers should follow. We have also learned how to refine the Hindsight design. Most importantly, we have learned that the reordering technique is a valuable tool for debugging real programs.

Future Debuggers

Let us look to the future. In May 1993, parallel debugging researchers held a conference, where they presented current work and speculated on future work. One participant, Professor Charlie McDowell, wondered if parallel languages that eliminate ordering errors—for example, High Performance Fortran—would obviate the need for work in this area. Some participants thought these languages surely would.

For at least the foreseeable future, ordering errors will exist. Though data parallel languages such as High Performance Fortran eliminate ordering errors, these languages are far from ubiquitous. These languages represent a tradeoff. They eliminate errors, but only by sacrificing performance. Since many programmers are unwilling to sacrifice performance, ordering errors are quite unlikely to disappear in the near future.

In fact, ordering errors are likely to become more common in the foreseeable future. Programmers searching for greater speed will try to divide their programs into smaller actions. Programmers will thus have to synchronize more actions, leaving more room for error. Smaller actions also mean more room for errors to hide.

Further, finding ordering errors will become even more difficult than it is today. With thousands of processors, each capable of running an action, the programmer will have a harder time knowing for certain what causes an error. He will want to try alternatives, implying that the reordering technique will be even more important in 2005, when many more large parallel machines will exist.

The sheer amount of log data from machines with thousands of processors will force in-

tegration of debugger features. Current research debuggers provide one or two features. No commercial debuggers include many features prevalent in the debugging literature. Frustrated customers will change that; they will demand integrated features in order to face the daunting errors in parallel programs. Programmers will want help.

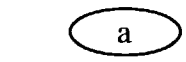
Computers, however, are not equipped for all tasks. They cannot find ordering errors alone. They cannot even find them when programmers limit the search space. Programs that find ordering errors work on very small programs only; they require too much computation to find errors in large programs. We cannot rely on computers to find errors. They can just point out possibilities to the programmer.

The programmer must bear the burden of fixing ordering errors. Once a programmer finds an apparent error, he must know that it will not vanish before he has a chance to fix it. Only the reordering technique provides this assurance. Thus, at the heart of integrated debuggers of the future will be an invaluable tool, the reordering technique.

Appendix A

Figure Key Compilation

The figure on the next page explains the shapes and patterns used in the figures throughout this thesis.

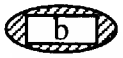


Atomic function execution named a

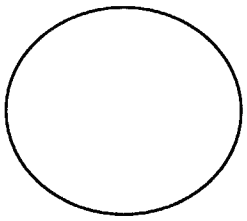
Message with identifier number 4

Switch point

Reordered function named r



Function execution, named b, that can happen in a different order than in one logged execution



Debugger component

The component at the origin of the arrow supplies information to the component pointed at by the arrow

Atomic function execution that happened between the send and reply of a remote procedure call

Graph node corresponding to function execution B

Edge in graph

One processor's copy of a B-tree node

Figure A-1: Compilation of figure keys.

Bibliography

[AP87] Todd R. Allen and David A. Padua. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the International Conference on Parallel Processing*, pages 721–727, 1987.

[Bal69] R. M. Balzer. EXDAMS - EXtendable Debugging And Monitoring System. In *AFIPS Proceedings. Spring, Joint Computer Conference*, pages 567–580, 1969.

[BDCW92] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 Conference*, June 1992.

[Ber91] Anton Beranek. Achieving Replay for Light-Weight Process Teams in the MOSKITO System. In *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 210–212, 1991.

[BM72] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[BN84] A.D. Birrel and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CKS90] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of Event Synchronization in a Parallel Programming Tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, Seattle WA, March 1990.

[Dav92] Al Davis. Mayfly: A General-Purpose, Scalable, Parallel Processing Architecture. *Lisp and Symbolic Computation: An International Journal*, 5(1/2):7–47, May 1992.

[DS90] Anne Dinning and Edith Schonberg. Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–10, March 1990.

[EP89] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):89–99, January 1989.

[GGLS91] Arthur P. Goldberg, Ajei Gopal, Andy Lowry, and Rob Strom. Restoring Consistent Global States of Distributed Computations. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging 1991*, pages 140–149, 1991.

[JC92] Ted Johnson and Adrian Colbrook. A Distributed Data-Balanced Dictionary Based on the B-link Tree. Technical Report MIT/LCS/TR-530, Massachusetts Institute of Technology, 1992. Also available in the IPPS '92, pp. 319–324.

[Knu73] D. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Scarching.* Addison-Wesley Publ. Co., Reading, MA, 1973.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Law76] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, 1976.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[LR85] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 515–522, Denver CO, May 1985.

[LS86] V. Lanin and D. Shasha. A Symmetric Concurrent B-Tree Algorithm. In *1986 Proceedings Fall Joint Computer Conference*, pages 380–386, November 1986.

[Mal92] Dalia Malki. Nicke — C Extensions for Programming on Distributed-Memory Machines. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, pages 103–118. Elsevier Science Publishers B. V., 1992.

[MC89a] John M. Mellor-Crummey. *Debugging and Analysis of Large-Scale Parallel Programs.* PhD thesis, University of Rochester, 1989.

[MC89b] Barton P. Miller and Jong-Deok Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 141–150, 1989.

[Net93] Robert Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[NM92] Robert Netzer and Barton Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *SuperComputing '92*, pages 502–511, 1992.

[Sch81] Robert David Schiffenbauer. Interactive Debugging in a Distributed Computation Environment. Technical Report MIT/LCS/TR-264, Massachusetts Institute of Technology, 1981.

[Sch89] Edith Schonberg. On-The-Fly Detection of Access Anomalies. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 285–295, July 1989.

[Smi84] Edward T. Smith. **Debugging Tools for Message-Based, Communicating Processes.** In *Proceedings of the Fourth International Conference on Distributed Computing Systems*, pages 303–310, 1984.

[Tay83] Richard N. Taylor. **A General-Purpose Algorithm for Analyzing Concurrent Programs.** *Communications of the ACM*, 26(5):362–376, May 1983.